Krists Kreics

Quality of analytics management of data pipelines for retail forecasting

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 29.07.2019

Thesis supervisor:

Prof. Hong-Linh Truong

Thesis advisors:

Dr.Sci. (Tech.) Mikko Ervasti

M.Sc. Teppo Luukkonen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Krists Kreics

Title: Quality of analytics management of data pipelines for retail forecasting

Date: 29.07.2019 Language: English Number of pages: 54+3

Degree programme: Master's Programme in ICT Innovation

Major: Data Science Code: SCI3095

Supervisor: Prof. Hong-Linh Truong

Advisors: Dr.Sci. (Tech.) Mikko Ervasti, M.Sc. Teppo Luukkonen

This thesis presents a framework for managing quality of analytics in data pipelines. The main research question of this thesis is the trade-off management between cost, time and data quality in retail forcasting. Generally this trade-off in data analytics is defined as quality of analytics.
The challenge is addressed by introducing a proof of concept framework that collects real time metrics about the data quality, resource consumption and other relevant metrics from tasks within a data pipeline. The data pipelines within the framework are developed using Apache Airflow that orchestrates Dockerized tasks. Different metrics of each task are monitored and stored to ElasticSearch. Cross-task communication is enabled by using an event driven architecture that utilizes a RabbitMQ as the message queue and custom consumer images written in python. With the help of these consumers the system can control the result with respect to quality of analytics.
Empirical testing of the final system with retail datasets showed that this approach can aid data science teams to provide better services on demand with bounded resources especially when dealing with big data.

Keywords: machine learning, offline learning, data pipelines,
quality of analytics, apache airflow

Acknowledgements

I would like to thank my wife Kamilla for supporting me in the long hours that went into this work and allowing me to fully focus on writing. I really cannot believe in the way we flow.

I would also like to thank my professor Hong-Linh Truong for providing interesting discussion points and feedback.

Finally I would like to thank my advisors Teppo Luukkonen and Mikko Ervasti and all the wonderful people at Sellforte who provided me with a fun and challenging working environment that truly was one of a kind.

Otaniemi, 29.07.2019

Krists Kreics

Contents

Abbreviations and Acronyms

API	Application programming interface
AWS	Amazon Web Services
DAG	Directed acyclic graph
ECS	Amazon Elastic Cloud Service
EC2	Amazon Elastic Compute Cloud
ML	Machine Learning
QoA	Quality of Analytics
SDK	Software development kit
S3	Amazon Simple Storage Service
vCPU	Virtual CPU

1 Introduction

With the rapid increase in computing power and the growth of available data machine learning has become a mature topic in the last decade [1]. Nowadays enterprises of different sizes use some kind of machine learning solution to either improve their business or enhance their product. The development and deployment of such solutions bring new overhead to these enterprises. A typical machine learning solution does not only consist of the program code that is used for training models but it also has to be able to connect to a training data source, store the trained models and make these models available for usage in different systems.

Typically machine learning systems are composed as data pipelines. A data pipeline orchestrates different data processing tasks. Usually, tasks are executed consecutively and the result of the pipeline is a function or model that is utilized to make predictions [2].

Offline learning is a subset of machine learning workflows [3]. Such workflows do not change the approximation of their target function once the initial training phase is done. This setting is usually utilized when there are infrequent data updates and when the whole dataset can be used for training the model or target function.

The first focal point of any machine learning solution is to develop an algorithm that can approximate the given data with good enough accuracy. After the algorithm is constructed evaluation of it is done. If it is good enough an optimization process can be started. For example, in startups, one might want to have control over different aspects at the same time. This is because requirements can change quickly and the resources for example time and money have to be utilized carefully. This topic has not been studied extensively and requires deeper examination and more practical applications.

Since data pipeline management is still a young topic the available literature and online resources are more limited than for more general software engineering topics. This is why the goal of this thesis is to assess the current state of the art practices and technologies and then based on that knowledge architect and develop a solution that provides trade-off management and quality assessment capabilities in an industrial setting.

1.1 Contributions

The main contribution of this thesis is a framework that gives control over trade-off management and provides task-level monitoring capabilities. The proof of concept is open-sourced and available at https://github.com/kristsellforte/qoa_framework. Currently, only a limited amount of practical examples for such pipelines exist. The framework was tested with retail data in collaboration with Sellforte [4], a Finnish startup specialized in marketing and promotion analytics. A secondary contribution of the thesis is a thorough comparison of existing machine learning frameworks and common workflows.

1.2 Structure of the thesis

This thesis is structured in nine sections. The first section introduces the thesis and highlights contributions. The second section provides a deeper background and sets out clear goals for the thesis. The third section features a literature overview that highlights current challenges in model management and approaches for managing different trade-offs in data pipelines. The fourth section features an overview of the current state of the art machine learning frameworks to provide a deeper understanding of the current industrial context. An overview of useful related technologies is done in the fifth section. The sixth section describes the architecture of the system. The system and empirical results are evaluated in section seven. Section eight features a general discussion about the solution. Lastly, the ninth section contains a conclusion that summarizes the work and highlights key learnings.

2 Background

This section provides the needed background information. This section is divided in four subsections. The first subsection describes the case company. The second subsection describes the problem and highlights research questions. The third subsection describes the main requirements for the system. The fourth subsection defines challenges related to forecasting in a retail setting.

2.1 Case company

To provide a better foundation for the work we first provide information about the setting the work was done in. The case for this thesis was provided by Sellforte Solutions Oy in the spring of 2019.

Sellforte [4] specializes in forecasting and optimizing sales and margin for B2C companies by analyzing their marketing and promotional activities. Currently, Sellforte works with leading retailers in the Nordics and are growing their analytics capabilities with an expansion to other industries and regions. Typically most of the analytics are done in an offline learning setting. The results and suggestions are presented to the clients via a proprietary web-based user interface.

As part of this expansion, Sellforte is introducing new features and analytics capabilities in the product in quick succession. The company is striving to become a leader in their industry and to do so, they are always looking into new technologies related to data engineering and data science that could improve their product. This thesis is part of that process and allows to explore different tools and methods related to managing data pipelines and related data engineering processes that would allow to optimize the usage of cloud resources and introduce a higher degree of control.

2.2 Problem statement

Sellforte and other companies working with data analytics are interested in providing the best possible data analytics results for minimal cost within a reasonable amount of time. In order to have control over these processes proper monitoring tools have to be in place. With the help of monitoring tools engineers and data scientists within a company can see how their analytics process and data pipelines can be improved to increase productivity and provide higher client satisfaction.

The thesis will focus on building a suitable system and analyzing different processes for developing data pipelines and testing various approaches for managing the common trade offs in machine learning and data science in general. We have set out three main research questions:

1. What is a suitable architecture for data pipelines in an offline learning setting?
2. How can one manage the trade offs between data quality, time, cost and other related metrics?
3. What are the benefits and costs for adding trade-off management capabilities to existing pipelines?

We can also define the process for acquiring answers to the questions. The first question will be answered by statically evaluating the current state of the art technologies. Suitable technologies and architecture will be selected based on that evaluation. Research of the second question consists of two parts. First, a literature study that allows us to discover different methods and processes related to trade-off management and second a practical implementation of different methods and their comparison. The third question will be researched by empirically running the final system in different settings and measuring its benefits on specific use cases as well as the involved cost or time overhead.

2.3 System requirements

This subsection defines the key features that are taken into account when designing the final system and evaluating existing solutions. The system that is being designed is a framework for data pipelines with trade-off management support.

The first requirement is that the system has to be a cloud provider agnostic. This means that it cannot be bound to one specific service provider such as Amazon Web Services or Google Cloud. Different use cases and different systems can require different resources and systems from the cloud provider and the provider itself can change. This means that the system has to be able to accommodate such changes so options for extending the framework have to be easy to use and changes in this respect should not affect the functionality of the system.

The second requirement is that the task structure has to be modular and support custom code. This means that the individual tasks can be deployed to different service providers and different instances that may vary in size. This is done to provide additional control over the elapsed time and cost for individual pipelines and tasks. An example where this is useful can be seen when defining two different tasks - a model training task that is both memory and CPU intensive and a data adjustment task that is only memory intensive. In each case, we should be able to provide different resources that fit the task at hand.

The third requirement is visual feedback of an ongoing process within a pipeline. To ensure that the data scientist controls the analytics process we have to provide tools that give feedback on the currently running tasks. The feedback should allow the data scientist to see metrics related to resource consumption in real-time as well as provide insight into the quality of the result. Looking from the resource perspective, metrics such as CPU and memory are usually provided by most monitoring systems but such low-level metrics are harder to justify when controlling the analytics process. This means that ways of defining high-level metrics have to be provided to the end-user of the system. All these metrics should be stored so that the end-user can compare how they have changed over time concerning the changes in the training algorithm or cleaning method. The presentation of the metrics should be visual so that these insights can be easily observed by various members of a data science team.

The fourth requirement is that the developed system has an interface that allows the data scientist to define a set of rules that stop the running pipeline if needed.

An example of the rule would be stopping the pipeline in case too much data rows were removed in the cleaning process, meaning that the resulting model would not be a good representation of the real situation. Such a feature automates part of the control process and allows the data scientist to focus on other tasks. Ideally, the same functions responsible for stopping the pipeline could also manage conditional branching depending on some metric. For example, if the data cleaning process took a long time and was very costly we would like to branch to a model training process that consumes fewer resources to analyze within the monetary limits specified by the user.

The fifth and final requirement is that we should also include a restrictable API. A restrictable API means that the endpoint is not publicly accessible and can only be accessible from a predefined range of IP addresses. Such a requirement ensures that sensitive information stays confidential and since sales data from retailers is sensitive information we should apply this precaution to our system. This fifth requirement does not have to be part of the framework itself as it does not concern trade-off management directly. The only limitation for this part is that the end-user has to be able to retrieve predictions in near-instant timing ($<$ 1 second).

The set of these main features will allow the system to be reusable and it will provide the data scientist with the essential tools needed to manage the trade-off between cost, time and quality of the result. It would suffice to only satisfy requirements three and four in order to create system capable of managing basic trade-off management but the goal is to fit these tools into a modernly designed data pipeline framework so that pipelines can be built in the same place and utilize the same tools.

2.4 Test case

The setting that will be examined during this thesis is one of offline time series forecasting for retail. In this setting, data updates happen infrequently and the update period might change so automatic retraining is not necessarily required but instead can be re-triggered manually.

Throughout this thesis, the details of the model will be treated as a black box. This allows us to focus on the engineering parts instead of model optimization.

The general trade-off that will be studied in the test case is one between computation power and the granularity of the prediction. The framework should allow the data scientist to bound one of these dimensions and interact with the pipeline to achieve a result within the predefined bound. Computation can be expressed as a mix of time and money. With more money, one can provision more cloud instances for running the computation and this takes less time. With more time one can do the computation on a lower cost instance for a long period. Full elasticity in this regard would be hard to achieve so it is better to fix one of these dimensions. Greater attention should be paid to the trade-off of quality and money since offline forecasting is less time-sensitive.

3 Literature review

This section consists of two subsections. In the first subsection, common challenges in machine learning pipeline management are discussed. The second subsection looks at the current state of quality of analytics and the challenges associated with it in data analysis settings.

3.1 Machine learning pipeline management

In most theoretical cases machine learning is treated as an ad-hoc solution for providing predictions on a specific metric given a static dataset [5]. In reality, it often is much more complex than that. The machine learning pipelines in production have to correctly handle model updating, result serving, user input validation and they have to provide feasible predictions with small errors all of this has to be delivered as continuous service with outage rates close to 0. Another aspect is that since these algorithms are usually computationally heavy they are run on the cloud which means that a typical machine learning production pipeline would consist of various cloud services [5].

Typically a machine learning pipeline consists of multiple steps that handle the retrieval of data, the cleaning of data, feature engineering, model training, model validation and the serving of predictions [2]. These steps are reflected in a simplified pipeline sketch in Figure 1. Since a high frequency of analysis has to be conducted in continuous service pipelines, the steps and interactions between them have to be automated. During this process, the developer has to make sure that in the transitions between the steps the dataset does not lose any valuable information [6]. For example, a situation could arise in which the dataset is mutated during execution by another data scientist and thus the result is not what was expected. Another important aspect that the developer has to take into account is that the each of the steps has to be developed in a modular way [6]. Since a situation could arise in which one step needs to be done completely differently. For example when the machine learning algorithm behind the model changes we have to update the step that takes care of the training. Another reason in favor of modular steps is that some of them could be reused in a different pipeline, for example, the data cleaning step could be reused across several pipelines [6].

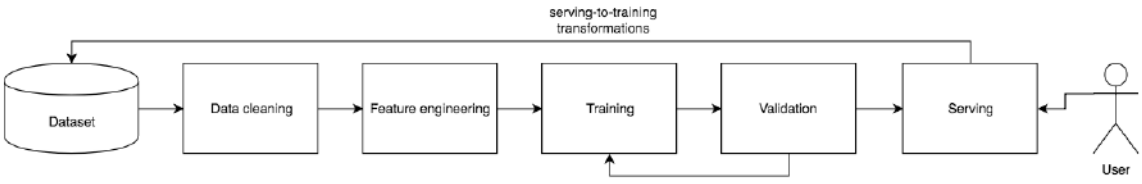

Figure 1: Simplified view of a machine learning pipeline in production

The pipeline sketch in Figure 1 contains a validation guard that branches out in two states. During the validation step, a developer would estimate the accuracy of

the model by using cross-validation [7]. The validation step should not mutate on data or model changes as the same metrics have to be used for evaluation over time if we want to compare the historical scores. These metrics have to be selected carefully and usually are specific for each algorithm. For example in time series forecasting for retail, the developer can split the sales data incorrectly so that an important feature such as seasonality is not reflected in the subset. It is also hard to attribute changes in scores to specific changes in the code when bigger changes are done, so the developer has to validate the model as often as possible by implementing small incremental changes [6, 7].

The previously mentioned training process actually should be repeated whenever significant changes in the data or code take place. This also brings additional challenges. The developer has to decide on how often should the models be retrained. For example, is the retraining done every time a new data point is added or can it be only done once in a predefined period? Some new data points can cause unnecessary noise in the data and negatively affect the performance of the model. This means that the process should be audited by a human from time to time to ensure the integrity of the models [6]. It is also recommended to keep versions of the models by storing the model and the specific combination of code and data that yields that model. With the help of versioning, the developer can rollback to a previous model if new data points have affected the performance [8].

There are many challenges associated with model management in machine learning pipelines and from the overview done in this section one can draw three bigger conclusions. Firstly, pipelines have to be modular and the developer has to design each step in the pipeline so that it can be reused across different setups if needed. Secondly, while code and data can change over time the underlying metrics that the developer uses must be predefined and immutable for comparison of different model versions. Last but not least while most tasks should be automated some tasks in the pipelines are challenging to automate and can require human intervention. An example of such a task is model retraining that requires changes to be made in the code that create a result which better approximates the data at hand.

3.2 Managing quality of analytics

Different data science applications have different needs, while some require optimization for the quality of results other applications might be made to optimize the performance or costs associated with getting the results. Typically different data management architectures focus only on controlling one dimension from the following - cost, performance or data quality. We want to look at the combination of these features and managing the trade-off between them. That is why we should look at the concept of quality of analytics.

In Ref. [9] Quality of Analytics (QoA) is defined as follows "We identify and define quality of analytics (QoA) in dynamic and diverse environments, e.g., based on cloud computing resources for big data sources, as a composition of quality of data (data quality), performance, and cost, to name just the main factors". Generally, this means that the management of quality of analytics deals with controlling the

trade-off between the cost, time, output quality or any additional metric of an analytics pipeline. A general example would be the task of finding the lowest price for a certain product. In this case, if we narrow the search only to a specific district within a city the computation will take less time and cost less, on the other hand, the result will only be a local maximum. Essentially a pipeline that is built with quality of analytics management in mind would support various levels of granularity depending on how much the end-user is willing to pay or wait.

One of the biggest challenges in quality of analytics arises from assuring data quality. The process of data quality management is complicated as it requires a collaborative effort from experts of different domains [10]. Commonly this would involve the domain expert, data scientist, and the engineer. So before implementing QoA in our pipelines relevant data quality metrics have to be defined and the developer needs to understand how changes in the input data and available resources may affect the outcome. Only after fully understanding data quality management we can move on to the management of QoA.

The metrics used for defining quality and measuring resources often can be different depending on the data and analytics use case. This leads to a need for a systematic way of defining how to develop primitives for data assessment and adjustment. One good approach would be to establish primitive action models. Primitive action models (PAMs) are defined in more detail in Ref. [11]. In general, a PAM performs data assessment and quality adjustment if needed. In our case, we could have an assessment action that calculates the error of a forecasting model. That action can trigger another action that retrains or drops the model if the error is too big. Generally, a primitive action can express various things. A primitive action can be an adjustment action, a monitoring action or a resource control action. Primitive actions would be defined in a separate file and executed through an invocation of an external service. In this thesis, all primitive actions will be treated either as adjustment actions or resource control actions.

Currently, research in quality of analytics is limited and there are not a lot of implementations that we could study. During the research for this thesis, only 4 relevant research papers that study quality of analytics were found in the IEEE and ACM libraries. There is at least one paper that shows a practical implementation of quality of analytics. That paper can be found in Ref. [12]. The focus of this paper is on quality of analytics management in an edge computing setup. Edge computing in a data analytics setup means that the data up to a certain extent is processed closer to the location where it is needed [13]. This improves response times and saves bandwidth. The trade-off is managed by aggregating data on the edge and minimizing the communication with the cloud while still achieving optimal quality. More specifically the edge nodes are enhanced to decide which data and when to deliver. Minimizing the communication with the cloud and thus also the computation is done in the cloud minimizes the total cost of the analytics. The decision to whether to communicate the data or not is done by sophisticated rules that are based on the knowledge about the algorithms behind the analytics. These rules are a good example of what quality of analytics management systems should support. The main novelty of Ref. [12] is that it shows the viability of quality of

analytics and provides us with a reference point for system development.

In the end, the goal of a successfully implemented quality of analytics system is not only the correctness of the prediction but also the systems ability to provide the data scientist with relevant tools to control and monitor the process to achieve the expected quality of analytics [9]. Often the analytics algorithms can be complicated and domain-specific and this thesis will not look at optimizations for specific algorithms or optimizations related to the analytics process itself.

4 Overview of existing frameworks

This section is divided into four subsections. In the first subsection, the relevant attributes for comparison are highlighted. In the second subsection, a comparison of the current state of the art machine learning frameworks is made depending on the defined attributes. In the third subsection, a summary of all the featured frameworks is made. Last but not least the fourth subsection features different frameworks that focus around experimentation in machine learning. The main goal of this section is to gather best practices for machine learning pipelines and understand if any of the currently existing solutions can be used as a foundation for the final system.

4.1 Comparison attributes

The evaluation focused around attributes that are necessary to create external services that could provide such control.

From quality of analytics perspective, we looked at three different attributes - performance monitoring, data monitoring, and cost structure. If an existing solution provides in-depth performance monitoring, data monitoring and has a clear cost structure we can define an external control service that uses the information from the framework and controls the process. These attributes are justified as follows.

A crucial part of quality of analytics is understanding and retrieving the metrics related to cost and time and these metrics are usually a function of the consumed resources so to provide any kind of QoA support within the pipelines we need to retrieve performance metrics.

Data monitoring is relevant as it directly corresponds to retrieving the actual quality of a result which is one of the cornerstones for QoA.

Cost is one of the relevant attributes that is taken into account when managing QoA, so to maximize quality per cost we need the solution to be either cheap or easily manageable.

On top of these three attributes, we added three attributes that correspond to the previously set out requirements for the system in section 2.3. These attributes are cloud agnosticism, custom model support, and restrictable API support. These attributes were already previously justified in the requirements section.

Together that gives six distinct attributes that will be used when evaluating the frameworks and comparing them with each other.

4.2 Machine learning frameworks

In this section an overview of seven common machine learning automation frameworks is made. The overview is made based on the previously defined six attributes.

4.2.1 AWS Sagemaker

AWS Sagemaker [14] is the machine learning pipeline from Amazon Web Services made for decreasing the complexity of dealing with machine learning in production. Models can be built on AWS hosted Jupyter Notebooks. The standard setup comes

with common machine learning libraries such as Pandas, Tensorflow and Numpy. Additional frameworks can be brought in from docker containers that are deployed on AWS EC2 (cloud computing) instances [15]. AWS Sagemaker utilizes S3 as the main storage for training data and resulting models, to speed up model training data can be pulled in from S3 on to a 5GB EBS Storage which is coupled with the Jupyter Notebook. During the training, data is pulled from an S3 endpoint and used on the built model. Automatic updates to the models are not provided but AWS provide guides on how to set up an AWS Lambda [16] that can trigger the start of the training process when data is updated in the S3 bucket. The Sagemaker training process provides automatic model tuning which works with the machine learning libraries supported by AWS, these libraries are TensorFlow, Apache MXNet, PyTorch, Chainer, Scikit-learn, SparkML, Horovod, Keras, and Gluon. AWS Sagemaker also provides model monitoring for Accuracy, MSE, and other general metrics. AWS CloudWatch can be used to monitor the resource consumption of the processes. The results are stored on a separate EC2 instance that provides HTTPS endpoints that accept POST requests and respond with the prediction coming from the supplied payload. Endpoint access can be restricted by using an Amazon VPC that allows defining security groups with specific IP ranges [14]. AWS Sagemaker is not cloud provider agnostic solution that can be used for different companies and pipelines since it is developed and maintained a cloud provider - Amazon Web Services. Another point worth mentioning is that the full advantages of using Sagemaker cannot be achieved for custom models.

4.2.2 Azure ML Service

Azure ML Service [17] is a machine learning SDK made for python. This means that Azure ML Service provides more flexibility than similar services from Google or Amazon. The SDK can be used to train models on your local machine. The SDK is framework and library agnostic meaning that the developer can build custom proprietary models if needed. When doing training on the cloud the data can be pulled in from either Azure Databricks, Azure Storage or Azure SQL so there is some flexibility in this aspect. While Azure has their monitoring solution Azure Monitor there are no mentions of its integration with the Azure ML Service instead ML Service focuses on model management and model monitoring. Azure ML Studio offers model management capabilities the SDK has methods that can be used to evaluate the accuracy and log inputs. Since Azure ML Studio also offers model updating then this is also monitored so the developer can view how additions to the dataset impacted the model. The resulting model can be deployed as a container image to five different compute targets. The most common one would be an Azure Kubernetes Service that exposes the model as a web service. As of 2019, the exposed endpoint will always be publicly available and there is no way of restricting the access range [17]. Azure ML Service is lightweight as it ships as an SDK which makes it a viable option for custom models but it still does not provide the necessary tools for monitoring and controlling the processes within a running pipeline and it is also not cloud provider agnostic.

4.2.3 Google ML Engine

Google ML Engine [18] is the machine learning solution provided by Google Cloud services. Google ML Engine by default supports Tensorflow, scikit-learn, and XGBoost. The custom containers which are meant for custom models are currently in Beta. Similarly, as AWS Sagemaker, Google ML Engine only supports one storage type which is Google Cloud Storage (equivalent to AWS S3). Training is done in the cloud using one of the specific training instances. The solution also provides automatic hyperparameter tuning for the default models. For large datasets, the data can be downloaded in the instances' storage to speed up the process. From the monitoring side, Google ML Engine offers full insights into CPU and memory consumption. These statistics can be broken down for each task. There are limited monitoring capabilities for custom models but for models using Tensorflow, there is a Tensorboard integration. Tensorboard is a visual tool that can be used for examining performance and debugging. The trained models can be stored as pickle files on Google Cloud Storage. The user has to specify the location of these models and the solution will take care of using those pickled files for making predictions on the data payload provided by some service. They also provide support to integrate with Apigee Edge. Apigee Edge provides private cloud solutions so the endpoint access can be restricted if needed [18]. Google ML Engine lacks support for custom models and is not cloud provider agnostic so the solution does not support all six attributes.

4.2.4 Pachyderm

Pachyderm [19] similarly to the other solutions to follow is not tightly coupled with any of the big cloud service companies. Pachyderm has no restrictions when it comes to the way models are being built and it supports any preferred language and framework. There are also no real restrictions for the storage and anything is allowed as long as the developer can manage to retrieve it in the codebase. This freedom comes with a trade-off as Pachyderm does not provide any solutions for monitoring on controlling the processes within a running pipeline. However, for resource monitoring on the cloud, the developer can use any of the solutions provided by the cloud service, e.g. AWS Cloudwatch, which do not provide task-level metrics. This is also mentioned on their Github [20]. To serve the results they need to be exported out of Pachyderm and served manually using an HTTP server. The biggest feature of Pachyderm is versioning for data and code. The versioning feature allows viewing the state of the data and code at a certain point of time as well as the output for that specific combination of the data and code [19]. This feature would be valuable and is definitely worth revisiting in further stages of development.

4.2.5 Apache PredictionIO

Apache PredictionIO [21] is an open-source machine learning server. PredictionIO is tailored for big data solutions supporting Hadoop but it does also have a Python SDK. PredictionIO has a template gallery that has templates with implementations

for common algorithms such as churn rate prediction and linear regression. Since it focuses on big data solutions PredictionIO uses Apache HBase. It also utilizes HFDS for staging data. This default storage can be swapped by using one of the 6 available data store backends. These backends also provide local file system and S3 storage support. A PredictionIO engine can be monitored and custom alerts can be set for when the engine uses too much memory or CPU. These events can also be logged to a remote API. PredictionIO's Event Server is built to continuously collect data from your application. A PredictionIO engine then builds predictive models with one or more algorithms using the data. Feature selection and online evaluation capabilities are available but they do not extend to custom models. There is also an experimental dashboard available but its use is not recommended in production environments. For result serving PredictionIO exposes endpoints that support dynamic queries in real-time once the solution is deployed as a web service [21]. Apache PredictionIO is a good solution for big datasets that utilize online training but does not fully satisfy the previously defined requirements.

4.2.6 Valohai

Valohai [22] is a machine learning management platform with a focus on deep learning. For model building, Valohai has suitable docker containers that support Tensorflow and Keras but since it utilizes Docker images the code can be anything built on a custom container utilizing any language from C to Python. Data for model training can be stored on any of the following: Amazon S3, Azure Blob Storage, Google Storage, and OpenStack Swift. The actions within the Valohai pipeline are called executions. Usually, model training is contained within an execution. Training can be automated by using of these executions but triggers for updating have to be set up manually. During the training execution, Valohai provides an overview of the progress which includes logs and metadata plots. Valohai has features for data and model monitoring. These are available via a user interface from which the developer can inspect the performance of the models and travel through the previous versions of the code and data combinations. While cost optimization and performance monitoring are mentioned on their landing page, the documentation does not mention anything about this. Result serving is done on the solutions' own private cloud. The predictions are exposed as HTTPS API endpoints. Deployments are also versioned meaning that at one point of time a client can have access to more than one version of the predictions [22]. Although Valohai brings the most value to deep learning models it packages some unique features that could be valuable thus could be revisited in the future. One of such features is visual process inspection.

4.2.7 Kubeflow

Kubeflow [23] is a machine learning kit for Kubernetes. Model building for Kubeflow is done within a Jupyter Notebook. The Jupyter Notebook can be used to build any kind of custom python model but the full advantages of Kubeflow are only supported for models utilizing Tensorflow. Kubeflow does not integrate with any data storage solutions by default which also means that there are no strict limitations

and in general the best approach would be to use any cloud object storage. Model training is done as a separate task that runs in a Docker container. There is no specific support for training and within the container, the developer can use whatever method works best for the selected use case. Kubeflow by default does not provide any insights about costs or resource consumption. But since it is built on top of separate Docker containers, the resource consumption of tasks can be retrieved by observing the resource distribution in the containers. For data monitoring, Kubeflow does not have any support but Kubeflow automatically manages Hyperparameter optimization via Katib. There is built-in support only for serving Tensorflow and Pytorch based models. For custom-built models, they provide an integration with Seldon which allows deploying any machine learning runtime that is packaged in a Docker container [23]. Kubeflow highlights various valuable ideas, for example treating each separate task as a Docker container. These ideas should be taken into account when building the final system.

4.3 Evaluation of machine learning frameworks

None of the reviewed solutions fully correspond to the defined attributes and thus will not be used as the foundation for the final system of this thesis. The solution that has most of the attributes would be Apache PredictionIO but using custom models with Apache PredictionIO decreases the overall benefit of using this solution as some of the features are only available for previously defined setups.

Solution	**Pricing**
AWS Sagemaker	Building from 0.05 USD/hour Training for the memory optimized instance from 0.15 USD/hour Deployment from 0.07 USD/hour
Azure ML Service	Pay as go price is 0.176 USD/hour.
Google ML Engine	Pay as go price is 0.6915 USD/hour.
Pachyderm	Open source. Instance for running on the cloud starts from 0.096 USD/hour.
Apache PredictionIO	Open source. Instance for running on the cloud starts from 0.096 USD/hour.
Valohai	Licence costs 649 USD/month. For this licence the developer also has to supply the instance so an additional 0.096 USD/hour have to be paid. For a month with 31 day the average hourly cost would be 0.968.
Kubeflow	Open source. Instance for running on the cloud starts from 0.096 USD/hour.

Table 1: ML Solution basic pricing

Custom algorithms is a very important requirement for innovative companies that build machine learning solutions. Only 2 out of 7 solutions support fully custom algorithms. These solutions are Pachyderm and Azure ML Service. Unfortunately, these two solutions come with a different set of issues for example Pachyderm lacks any kind of monitoring support while Azure ML Service is not cloud-agnostic. The

comparison table that highlights the support for 5 out of the 6 defined attributes can be seen in 2.

Another attribute that was evaluated cost. To make the overview simpler the costs are evaluated separately here. Typically data pipeline tasks consume a lot of memory comparing to the CPU consumption, so a typical task would utilize a memory-optimized computation instance. Usually, such an instance starts from 0.096 USD/hour, depending on the provider [24]. The solutions created by cloud providers generally yield a higher base price since there is no option of running the solution outside of the solution creators infrastructure. This cost comes with its benefits as developers can save valuable time on deployment time and maintenance. Table 1 holds a list of the basic costs for each solution.

To conclude the evaluation we can state that none of the reviewed solutions completely fit the specified requirements. But some good architectural ideas and features could be featured in the final system of the thesis.

4.4 Experiment management frameworks

Experiment management frameworks are a subset of data science frameworks that emphasize model versioning and experiment reproducibility. The characteristics of these frameworks usually mean that they are not the best for continuous use in production environments. This is why in this thesis they are treated as a separate subset of tools and not mixed with more traditional machine learning frameworks. This also means they were not evaluated in the same matter. The only attribute we looked at here was QoA readiness, which defines if the solution has embedded QoA tools or not. In this subsection, an overview of two experiment management frameworks is done.

4.4.1 DVC

DVC also is known as Data Science Version Control is a model versioning tool that allows keeping track of model versions and the data and code combination that yielded that model. It is built to work alongside git meaning that a developer is supposed to push the analytics code to git and then DVC provides the opportunity to reference and link that commit with specific data input and a corresponding result. DVC position themselves as a tool for researching and looking for a great model (and pipeline), not optimizing and monitoring an existing one. They recommend that the result of this process is later incorporated into a data engineering pipeline such as Luigi or Airflow [25]. Essentially this means that by using DVC we would have to manage two separate pipelines - one for experiments and one for production. While tightly coupled versioning is a novel feature it does not allow us to control quality of analytics on running processes.

4.4.2 Polyaxon

Polyaxon is not as lightweight as DVC and offers a framework for experimentation and model management. Polyaxon similarly as DVC stores all data related to a

	Custom model support	Cloud agnostic	Performance monitoring	Data monitoring	Restrictable API access
AWS Sagemaker	Models can be built with any language or library but they cannot utilize all the benefits of the solution	No	Monitoring can be added as an external service from the provider	Inspection of general metrics possible	Can be restricted
Azure ML Service	SDK supports any language or library	No	Monitoring can be added as an external service from the provider	Inspection of general metrics possible	Cannot be restricted
Google ML Engine	Custom models only in Beta	No	Built in resource monitoring capabilities	Metrics only provided for Tensorflow models	Can be restricted with an external service
Pachyderm	SDK supports any language or library	Yes	No built in monitoring	No built in monitoring	No API support
Apache PredictionIO	Models can be built with any language or library but they cannot utilize all the benefits of the solution	Yes	Built in resource monitoring capabilities	Not available for custom models	Can be restricted with an external service
Valohai	Models can be built with any language or library but they cannot utilize all the benefits of the solution	Yes	No built in monitoring	Inspection of general metrics possible	Cannot be restricted
Kubeflow	Models can be built with any language or library but they cannot utilize all the benefits of the solution	Yes	No built in monitoring	No built in monitoring	Can be restricted with an external service

Figure 2: Table comparison of existing machine learning frameworks

specific experiment or pipeline run on the cloud and allows the data scientists to examine different aspects of each experiment so that they can make better decisions and optimize the model. On top of these main features, Polyaxon acts as a DevOps layer that manages deployments and automatically scales the available resources to improve performance. It is also worth mentioning that Polyaxon is cloud provider and language agnostic which means that it supports custom analytics algorithms that can be deployed anywhere [26]. All in all, Polyaxon is a framework that is suited for experimentation, data governance, and resource optimization. From all the frameworks that were assessed in the thesis, Polyaxon is the only one that not only strives for model optimization but also manages resource consumption but it does not provide any tools that allow to manage quality of analytics in the pipelines

4.4.3 Summary

While experiment management frameworks can be useful for larger data science teams in our case they would just add additional overhead to the rolling costs and time spent on pipeline management. Our final solution has to be a production scale pipeline that can be monitored and controlled in real-time with the respect to quality of analytics. So none of the experiment management frameworks will be involved in the solution.

5 Overview of related technologies

As discussed in the previous section none of the existing machine learning frameworks fully satisfy the requirements of our case. This means that there is a need to implement a solution of our own. This section looks at related technologies.

5.1 Task orchestration

As seen previously a typical machine learning workflow consists of various tasks, for example, data cleaning, feature extraction, and model training. These tasks have to be orchestrated so that they execute concurrently and only when all the data dependencies are satisfied. Ideally, there should be a way to automatically trigger different processes within the pipeline based on the results of different retrieved metrics. That is why in this section we look at 4 common task orchestration tools and evaluate them based on their features. The focus when doing the evaluation will be on task monitoring, the flexibility of what a task can do and abilities to control a running pipeline process. It is important to mention that overview of Apache Oozie and Azkaban is not featured in this section as these tools are more suited for Hadoop based workflows while we are in search of primarily a python based workflow.

5.1.1 AWS Step Functions

AWS Step Functions [27] is the workflow orchestration tool provided by AWS. Since the tool is closed source no implementation details about the system are known. Currently, AWS Step Functions can only orchestrate tasks that are written as AWS Lambda functions. AWS also provides the developer with a user interface that shows the status of an execution graph and highlights relevant details in case of errors. A valuable additional feature provided by AWS is state management. The state management makes sure that the state of the dataset is saved between tasks so a developer can later inspect the changes in the state throughout the workflow. Controlling a running pipeline that is built with AWS Step Functions can prove challenging as the in built mechanism for pausing a process requires additional waiting steps to be included in between the tasks. This means that the control condition can only be checked after a task has already executed which adds additional overhead and does not allow to stop the process if some resource related metric is exceptionally large [27]. Monitoring can be achieved by using AWS Cloudwatch but the results will not be fully real time.

5.1.2 Luigi

Luigi [28] is an open-source workflow orchestration tool originally developed by Spotify. The tasks in a Luigi workflow are organized in a dependency graph. A task typically ends up pushing the result to a *Target* from which another *Task* can get the data. A task waits for the dependency tasks to clear and those tasks are defined in a `require` statement so when a task is changed we also have to change the require statement in all the tasks that depended on it. The coupling of tasks and

data makes Luigi less modular meaning that tasks down the stream will depend on previous tasks. Luigi features a basic user interface that the developer can use to link and monitor tasks. Luigi also has a REST API interface that can be used to trigger or stop tasks via HTTP requests. It also should be added that the documentation does not provide enough details about usage to the developer. Last but not least Luigi is unable to do distributed execution meaning that the nodes do not scale well [28].

5.1.3 Netflix Conductor

Netflix Conductor [29] is an open-source orchestration framework for microservice management originally developed by Netflix. Since the framework is targeted at orchestrating microservices the worker tasks have to be provided as endpoints that can be executed over HTTP. Currently, Conductor does not execute user-supplied tasks [29]. The user interface of Netflix Conductor is basic and it provides almost the same monitoring functionalities as Luigi. All in all Netflix Conductor has capabilities that are a subset of those of Spotify Luigi.

5.1.4 Apache Airflow

Apache airflow [30] is a platform to programmatically author, schedule and monitor workflows. Apache Airflow represents workflows as directed acyclic graphs (DAGs). The nodes within such a DAG describe the tasks and each node can be any type of action, while the vertices describe the dependencies between tasks. The tasks are flexible and can be either a bash script, source code that does an ETL process or even an executable that lives inside its own Docker container. Tasks in Apache Airflow are atomic meaning that they are not coupled and do not share any direct dependencies. Apache Airflow also has an experimental REST API that exposes endpoints which can retrieve information about the running tasks and pause the execution of a pipeline [30]. Similarly, as the frameworks discussed before Apache Airflow features an user interface that can be used to monitor the running tasks and see the emitted logs.

5.2 Task organization

Typically tasks in data pipelines are just packages of source code. In our case that would be python source code. That source code can be stored in an instance that also manages the task orchestration with one of the tools mentioned in the previous section. Such an approach is quick and easy to deploy but not fully modular as all the tasks use the same instance and are part of the same image.

Overall we can derive three bigger strategies for task deployment. Tasks can be deployed as services that are accessible over HTTP [31], as program code on the orchestration machine or as a Docker image [32]. In an ideal scenario, the final system would support all three of the approaches but in reality, this might not be possible because of time constraints.

5.3 Data storage

Concerning data storage, we have set out two requirements. The storage must be protected from malicious activity and it has to be able to consume different formats of data since different companies have differently structured data.

AWS S3 [33] is a cloud-based object storage service provided by Amazon Web Services that offers scalability, data availability, security, and performance. There are certain advantages to utilizing S3 or similar object storage. It is scalable as there is no upper bound to the amount of data it can hold [34]. Also, the access to the storage can be limited with inbuilt tools. Object storages such as S3 provide an API that can be used to access files from any runtime [34]. The downside of using S3 is considerably slower query time because to access a certain item within a file or within the S3 bucket, one would need to traverse the tree of all files and the traverse the file itself. To mitigate this issue simplified sharding can be implemented that splits larger files into smaller buckets that are indexed by some hash function.

An opposite option from file system based storage would be to utilize a database (SQL or NoSQL). These options would require a separate database instance running that increases the total cost of running the system.

5.4 Model serving

Typically models are trained in a separate pipeline and then stored in the cloud at remotely accessible storage. This means that the serving part is decoupled from the overall pipeline. In the end, the serving module only needs to take care of retrieving the correct model from storage and applying incoming arguments to the model to return a prediction. This is not a computationally heavy task and it would only need 3-4 endpoints that provide the necessary information (available models and their predictions) to the client. The endpoints should be designed with elasticity in mind. Typically cloud services are elastic enough to scale up with the increase in usage so we look at cloud options exclusively. The biggest architectural decision that has to be made is whether to use a monolithic architecture based on a cloud computing instance or to use a microservice architecture based on serverless functions. In this subsection, we look at both approaches and compare the advantages of each approach.

A side by side overview of both approaches can be seen in Figure 3. On the left-hand side of the figure, we see an example of monolithic architecture. When using a monolithic architecture for serving models the endpoints would be hosted on one centralized instance. That instance has a runtime that executes the function corresponding to each endpoint and the same runtime has access to file storage which is usually located on the disk of the same instance. This means that all the functions can be overloaded and made unusable by just overloading one of the endpoints. When using serverless microservices each endpoint is hosted on virtually separated instances. These instances interact with remote file storage. With this approach, other functions are made available even if one of them is overloaded.

The first approach would be to use a monolithic web application framework

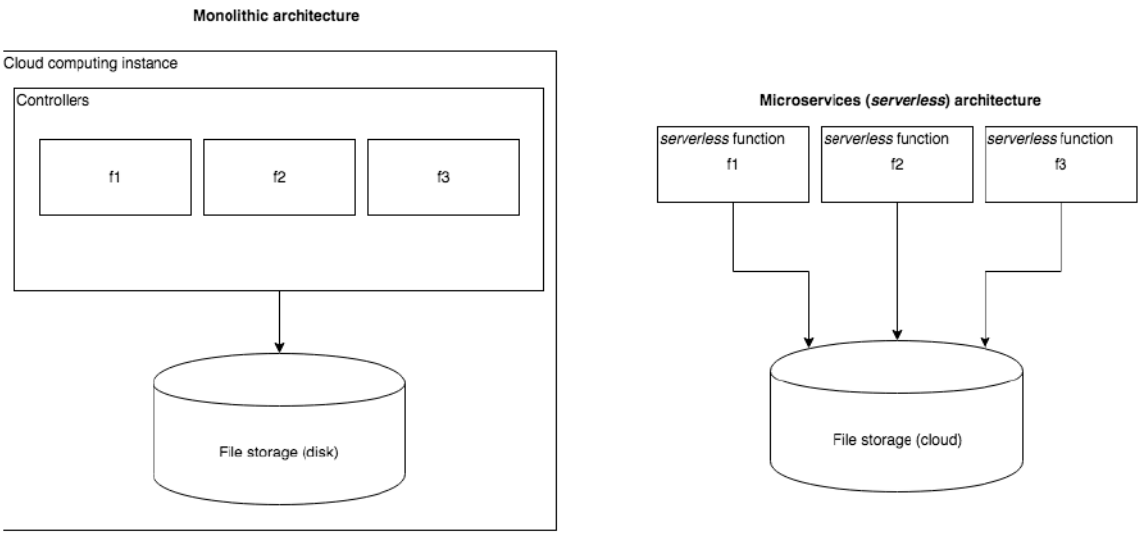

Figure 3: Serverless and monolith architecture sketches

that supports python such as Flask [35] and develop the logic that can serve the prediction demanded by the client. Flask would be the choice with the lowest development overhead since it is easy to wrap the developed prediction functions within a Flask endpoint. Given that the prediction functions are developed and fully functional set up the Flask application could be done in a matter of hours. The application could be tested locally and the models could be stored in the file storage of the application. If the application performs as expected locally it can be deployed to a cloud computing instance just by moving all the files to that instance. Additionally, we would have to install the necessary python packages on the remote server.

The second approach would be to use the so-called serverless functions in a microservice setting. Serverless functions are stateless functions that are hosted on the cloud and available as publicly exposed HTTP endpoints. Serverless functions promise higher security and scalability but they come with the trade-off in the form of limits on memory and computation time. For AWS Lambda the computation time per function call is limited to 15 minutes seconds, the disk space is limited to 500MB and the memory of instance can be in the range of [128MB; 3008MB] [36]. It is highly unlikely that such limits could be exceeded by a function that retrieves relatively small files ($<$ 1MB) from some object storage. There is another limit on the package size, so the package has to be under 50MB [36]. This limit can affect the use case for serving pickled models as common machine learning libraries as Tensorflow [37] and Facebook Prophet [38] exceed it so a workaround is needed in this case. To make the predictions accessible to clients each function has to be wrapped in a separate lambda and deployed as a separate HTTP endpoint. The predictions can be then tested by sending valid test inputs as HTTP POST requests to the generated endpoint. If code or functionality sharing is needed it is recommended to use one of the orchestration frameworks that also allows to set

up and test the functions locally. The framework with most features and direct integration with AWS is Serverless [39]. The setup and testing, in this case, would be time-consuming. Since the serverless functions go to sleep after being idle for 15 minutes so it is required to have a cron job that pings the function every 15 minutes to keep it *warm*.

5.5 Resource monitoring

To have a full picture of what is going on in our workflows we would need to introduce some type of resource monitoring. The tool used has to provide basic information about the resource consumption during a task and we have to be able to log that result in persistent storage. A separate service then would retrieve the results from that storage and visualize them to the developer. In this subsection, we look at different monitoring tools that support process monitoring. We did not look at monitoring tools provided by cloud platforms since by definition such tools are not cloud platform agnostic and would limit the final solution.

5.5.1 Cadvisor

Cadvisor [40] is a centralized monitoring solution for containers developed by Google. It provides an abstraction layer for the Docker Stats API and comes with a user interface that shows the relevant performance statistics [40]. Cadvisor was empirically tested with a docker-compose setup and what these tests revealed is that when using docker-compose Cadvisor can only observe the high level docker container process that includes all the running services and specific task or service level data cannot be retrieved with this tool which is an essential requirement to enable quality of analytics management.

5.5.2 Docker Stats API

For Docker images, Docker itself has an API [41] that allows the user to log real-time statistics about the running process into the standard output of a terminal REF. By using a bash script those same results can be manipulated and logged onto a file which is later uploaded to an AWS S3 bucket.

5.5.3 Psutil

Psutil [42] is a simple python library that retrieves statistics about the running resources. Since psutil is based on python we can wrap it in functions that augment the data in whatever format is needed for the management of specific trade-offs. These functions can then be used directly in the analytics scripts or ran as separate scripts in parallel to the analytics scripts if those scripts do not utilize python.

6 Technical solution

This section describes the created technical solution. In the first subsection, a high-level architectural overview is done. The second subsection goes a level deeper and describes the general API for using the custom components.

6.1 Architecture overview

This subsection contains an overview of the proposed architecture and shows how this architecture allows introducing the features defined previously. The section is further divided into five subsections each of them discusses one specific aspect of the framework. The first three subsections refer to background components that power the data pipelines. These background components are an Apache Airflow task runner, a RabbitMQ [43] message queue, Amazon S3 object storage buckets, a prediction API hosted on Amazon Lambda, an ElasticSearch [44] database with the connected Kibana [45] UI. These background components provide the foundation for the data pipelines. The latter two sections describe the main components used for managing quality of analytics. These custom components are written in python and directly deal with monitoring and controlling the trade-offs with respect to quality of analytics.

In Figure 5 we can see the visual representation of the high-level architecture of our system. In this figure, the main custom components are bolded. Throughout this section, additional references to this figure will be made.

Generally, the components of the system are composed as Docker containers that are orchestrated in one system with the help of Docker Compose. Such an approach allows to set up the system on any system both locally and on the cloud meaning that for the most part, the framework is agnostic to operating systems or cloud providers which is exactly what we defined as a requirement before. This approach also allows scaling the framework as images can either be deployed on a common instance or deployed on individual instances. In production, it is recommended that such a system is fully set up within a virtual private cloud that disables public access to the components. The only customer-facing components that need an interface to the public internet are Kibana and the prediction REST API.

6.1.1 Task runner

Based on the technology overview done in the previous section Apache Airflow was selected as the solution for running and orchestrating tasks. The main advantages for Apache Airflow that make it suitable are as follows. Firstly, it does not have strict limits for what a task can be. Secondly, it provides the most detailed user interface that allows monitoring of the task-related logs within a workflow. Thirdly, it has a REST API that can be used to control the process externally. Lastly, it allows for a completely modular workflow as the tasks are atomic and not coupled with respect to data or other upstream tasks.

Airflow itself consists of various components that ensure correct execution of the tasks within a pipeline. These components are colored in blue in Figure 5. Each of

the components mentioned here can be directly traced in Figure 5. First of all, we have the Airflow master node that runs the Airflow webserver which gives us a user interface that allows interactions with the pipelines and tasks within them. The same master node also runs the Airflow scheduler. The scheduler schedules tasks within a queue that is stored on a Redis store. Both the scheduler and webserver interact with the Airflow metastore. This metastore stores general information that can be reused across the scheduler and webserver such as credentials that allow connecting to outside services. The tasks within an Airflow pipeline are run using a separate worker node. The worker node picks up scheduled tasks from the Redis store and executes them. Tasks or operators within Airflow are atomic, meaning that they stand on their own and do not share any resources with other tasks [46]. In the solution, the worker node is a Celery worker. While airflow supports various worker node types Celery is the easiest to scale up when more tasks are added to the pipelines [47].

6.1.2 Data storage

Based on the technology overview done in the previous section Amazon S3 was selected as the solution for data storage. There are two advantages for Amazon S3 or other object storages when comparing to other systems. Firstly, Amazon S3 is almost infinitely scalable as there is no upper bound to the amount of data it can hold [34]. Secondly, Amazon S3 is cheaper as costs is only attributed to reading and writing operations and a separate continuously running instance is not needed. Thirdly, Amazon S3 has no limitation for the file format so raw data can be stored and retrieved directly with no preprocessing steps involved.

As already mentioned before Airflow tasks are atomic meaning that they usually do not share any resources. This means that there is a need to centralize the data storage so that all tasks within the same pipeline can interact with the same data. To do this we set up two separate Amazon S3 storage buckets. These buckets are represented as the green cylinders on the left-hand side in Figure 5. One bucket is used for data and the other for model storage. Usually, all tasks within a pipeline would interact with the data storage by taking some dataset and performing a transformation or computation with that dataset. The model storage is only used by model training and result evaluation tasks as well as the prediction API. This part of the system is coupled to the AWS services but similar services exist on different service providers and our system should be able to accommodate the change in providers.

The Airflow worker also interacts with a RabbitMQ message queue. The worker relays metrics that are related to quality of analytics to this message queue. This interaction will be described in detail later in this section.

6.1.3 Model serving

For model serving the serverless approach was selected. There are two main advantages to this approach. Firstly, this solution is cheaper, especially when the endpoints are not under heavy load. In some settings, AWS Lambda is up to 57%

cheaper and 1.66 times faster [48]. Secondly, this solution scales better since each endpoint can be scaled separately and on-demand without making any changes to the function code. These advantages come with a trade-off. The serverless approach requires additional orchestration by an external framework which means that there is more overhead associated with setting up the solution. It also has memory limits that will need additional work to be resolved. Serverless functions are also harder to debug than monolithic applications [49]. Overall this decision means that model serving will be optimized for cost and scalability.

In Figure 5 the model serving solution is represented as the white cloud on the left-hand side. The function itself is hosted on the cloud. The function is mapped to an HTTP endpoint. When that endpoint is queried with some parameters the function is executed. The function itself would usually retrieve a pickled model from object storage and apply the received parameters to it to retrieve a result.

6.1.4 Message consumers

In our system, we have two different consumers listening to messages added to the queue. These consumers are marked in white in Figure 5. On the left, we can see the ElasticSearch consumer. This is a custom consumer component written in Python that listens to new messages. All the messages posted on the queue are JSON formatted metrics associated with quality of analytics. The ElasticSearch consumer picks up a message and extracts the index information from it. Afterward, it stores the message in the relevant index on the ElasticSearch store. Data scientists can later interact with the indexes and visualize the metrics by using the connected Kibana UI. An example dashboard that was used when evaluating the system can be seen in Figure 4.

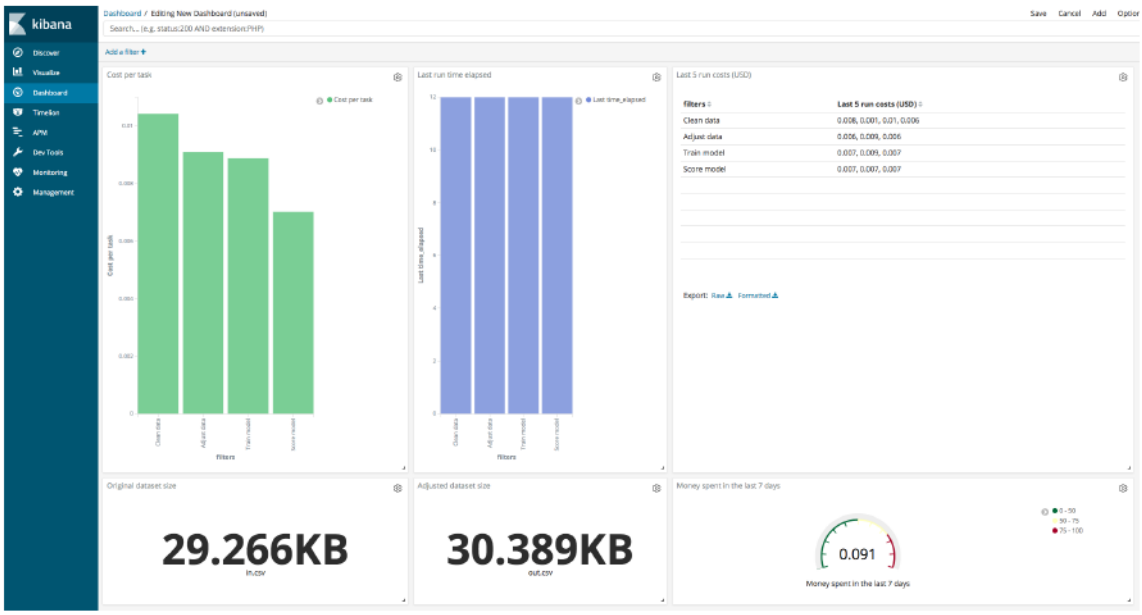

Figure 4: Example Kibana dashboard

On the right side of Figure 5 sits the Control consumer. The control consumer picks up messages from the queue and based on rules defined by the data scientist it either pauses or stops the pipeline. Pausing a pipeline means that the currently running task is finished but after that, the execution stops. The pausing can be done via the REST API provided by the Airflow web server which has a specific endpoint for pause. Stopping the pipeline means halting the execution of the current task and all the tasks following after it. The Airflow web server does not have an endpoint to do this so we have to interact with the Celery worker directly and stop its execution. Such stopping can introduce inconsistencies in the state of our pipelines and thus should be only used as a last resort. Both of the consumers discussed here are custom and directly deal with trade-off management with respect to quality of analytics.

6.1.5 Task design

To simplify the architecture pictures the design of an individual task has been separated from the high-level visualization and it can be seen in Figure 6.

Tasks within the framework are organized as Docker containers. The original idea was taken from one of the reviewed frameworks - Kubeflow. This design choice was made to make the tasks modular. With such an approach the task is isolated from the outside execution and allows to observe the resources consumed by each task individually. The isolation allows us to provide insights to each separate task and gives the data scientist a better understanding of which pieces within the pipeline are the most costly and depending on that the data scientist can decide to use a different task or improve the existing one. Another benefit of using Docker containers is that it allows us to limit the resources used by a task both locally and on the cloud. The resources locally are limited by provisioning less memory or CPU to the container and on the cloud the resource consumption is limited by deploying the task containers to appropriately sized instances. With this setup different tasks can be deployed to different service providers. Most service providers nowadays have container registries and container services that accept Docker containers so this design choice makes the tasks service provider agnostic and one could even use more than one service provider in a single pipeline.

The task containers also contain a custom performance monitor package that is written in Python by utilizing the Psutil library. This is the key component responsible for retrieving the correct metrics used for trade-off management. Psutil was chosen as the monitoring solution. Psutil is a python library which means that we can wrap it into a custom python class. This wrapper calculates higher level metrics such as cost on the fly from the received performance metrics. The same class can be extened to calculate different metrics depending on the needs of the user. There are two main advantages for Psutil when comparing to other tools. First it can be easily extended by using an expressive and popular language - Python. Second it is not tied to a specific solution, if needed the final package can be called from a bash script on any interface.

On startup the Docker container would usually run two python executables. One is the performance monitoring package described beofre. This package is run on a

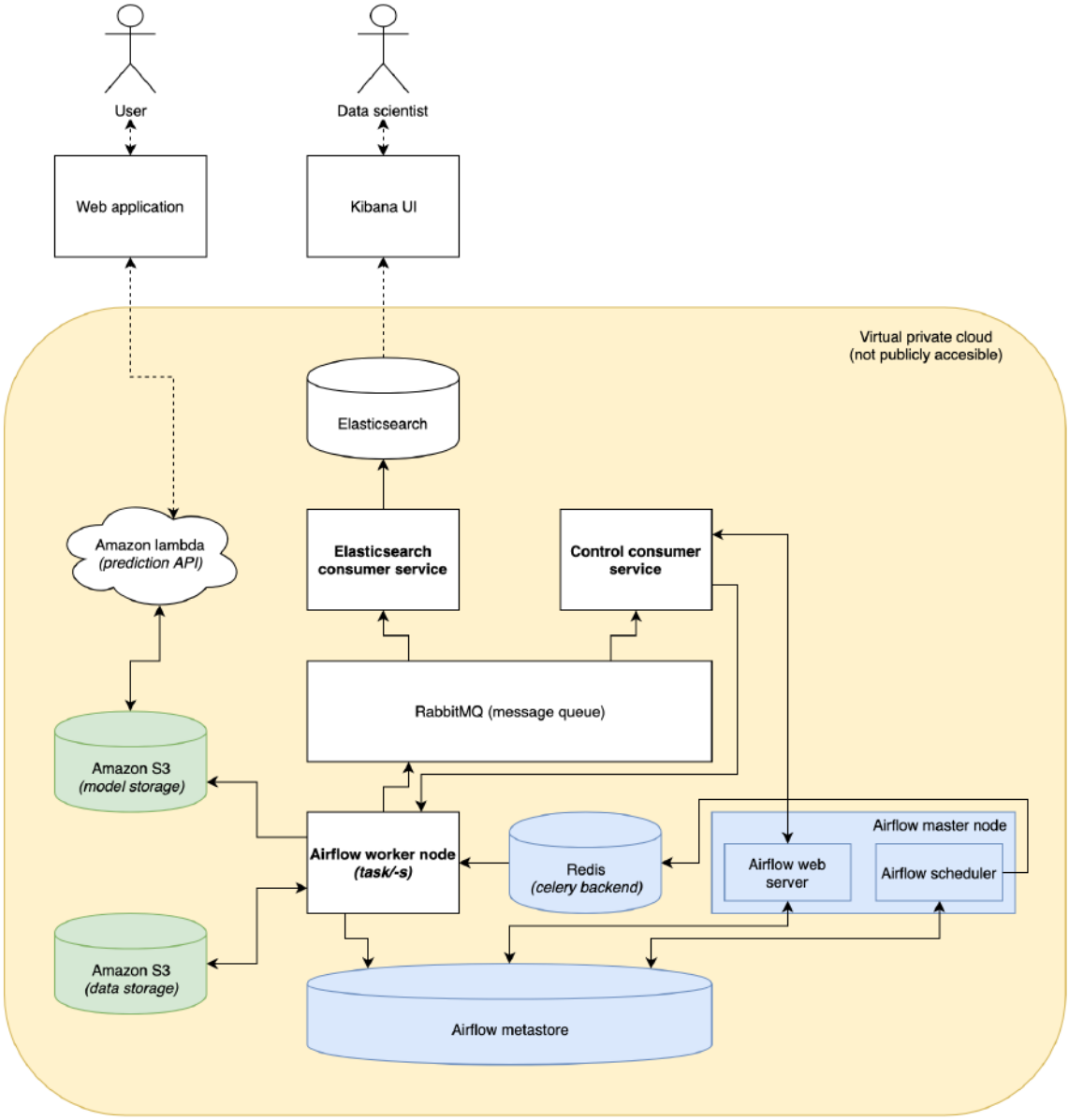

Figure 5: General architecture

separate thread and observes changes in resource consumption from the startup until the task finishes. All the collected metrics are relayed as JSON strings on an external RabbitMQ message queue. The other executable contains the logic of the task itself and interacts directly with the data and model storages. While the storages are marked as Amazon S3 buckets in Figure 6 generally the pipeline should be able to accept different types of storages.

To introduce conditional branching within the pipelines we can use the `Branch Python Operator` for the task. In this operator, we would need to retrieve metrics from the ElasticSearch indexes and direct the pipeline to the correct branch based on

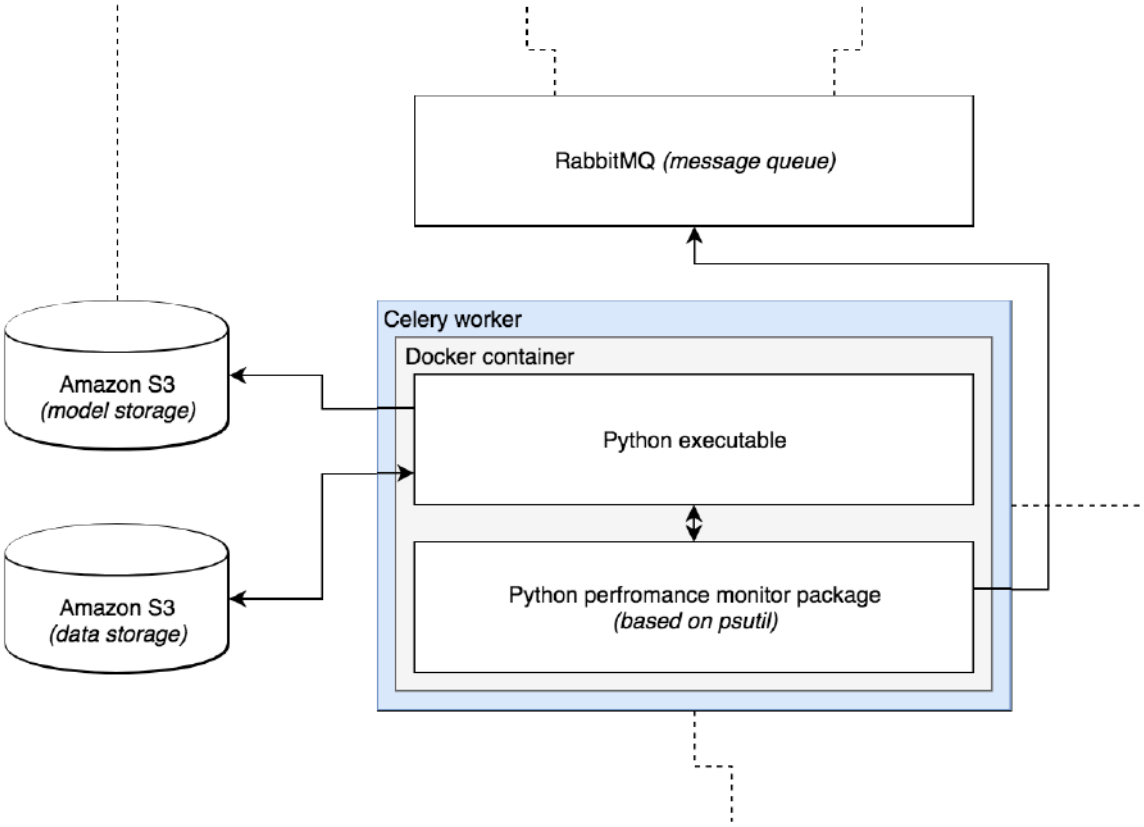

Figure 6: Individual task design

the received metrics. Empirically in this thesis branching will be done manually by triggering different branches in the DAG definition. It can be automated by storing the branching actions on centralized storage accessible by Airflow, for example, RabbitMQ or Amazon S3.

6.2 Using the framework

Since different teams can have different requirements and different algorithms are measured differently we have to make the API of the framework generic enough so that custom metrics can be defined and utilized. The framework itself provides out of the box tools that retrieve low-level metrics and it provides tools that can control the running process. To actually achieve meaningful management of quality of analytics one has to define higher-level metrics with the help of the custom functions. In this section, we will go over examples for defining a custom cost function, pushing custom metrics as well as setting custom control rules. These examples will help us illustrate the capabilites of the framework and showcase its usage. In this section, set up instructions are also included so that it is clear how pipelines are defined in our framework. Last but not least this section touches upon defining a different storage solution. Some of the explanations in this subsection require code samples so they are represented as figures where needed. More relevant program code to

this section can be found in the Appendix and the full solution source code is available at https://github.com/kristsellforte/qoa_framework. Through this section paths from the repository root will be referenced to provide a clear mapping between the files and the text.

6.2.1 Setup

The setup instructions are defined in the repository. For the framework the only strict system requirement is that the user has Docker pre-installed on their workstation. The framework has to be initialized by running an initialization script seen at `scripts/initialize_framework.sh`. This script will build the necessary images. When the images are built the user can start adding tasks. Tasks are added by a script seen in `scripts/create_task.sh`. When a task is added a new folder with the necessary requirements and Docker file is created. This folder contains a *code.py* file. The logic for the task is added in that file. The performance monitor is already initialized at the start of the same file. After all the necessary tasks are defined they can be added in the pipeline by defining them in the Apache Airflow dag file provided in `airflow/dags/pipeline.py`. The pipeline can be run by running `docker-compose up` from the root of the repository.

By correctly following the steps in the repository which we also briefly described here one can create a running pipeline that gathers and stores low-level metrics. But in order to enable quality of analytics, we have to define the previously mentioned custom functions. These definitions will be explored in the following subsections.

6.2.2 Setting a custom cost function

Since tasks are often run in different environments that have different pricing, we provide a way to define custom cost functions. Most common cloud providers define cost as a composition of resources and time, so we can utilize those metrics for calculating cost or similar higher level metrics.

An example cost function would be the AWS ECS Fargate cost function. This function was used as the main cost function in this thesis since the tasks essentially would be shipped as AWS ECS Fargate instances. The pricing for these instances is USD 0.04048 per hour for 1 vCPU. And USD 0.004445 for 1GB of RAM [50]. The definition of an AWS vCPU can change over time and can even change depending on the instance in used but for Fargate we can assume that the vCPU is a virtual core with a clock speed of 2.3 GHz [51]. Fargate also allows to use less than a full vCPU which means that the clock speed of the virtual core can scale up to 4 times lower to 0.575 GHz.

By default the cost function is ran once every second. Usually the pricing is defined per hour. So to show the cost increase in real time on a second per second basis we need calculate what is the cost per second. For the Fargate cost function we can just take 1/3600 of the given cost per hour.

After we have determined the pricing per second we can use it in our cost function. The function signature that can be seen in Figure 7 accepts 4 arguments.

First the `cpu` object retrieved by psutil, which contains information about cpu usage in that particular slice of time. Second the `ram` object retrieved by psutil, which contains information about ram usage in that particular slice of time. Third is the `elapsed_time` which is the elapsed time in seconds since starting the task. Fourth is the `previous_result` that contains the result of the function at $t-1$ the results from the previous call are usually added to the result of the current call to show the full cost since the start of the task.

```python
# cost function signature
def get_cost_function_object(cpu, ram, elapsed_time, previous_result):
```

Figure 7: Cost function signature in python

By using these arguments we can calculate the cost of the task in each function call. An example calculation for the AWS ECS Fargate cost function can be seen in Figure 8.

```python
def get_fargate_metrics_object(cpu, ram, elapsed_time, previous_result):
    # Fargate service cost per second
    FARGATE_CPU_COST = 0.04048 / 60 / 60
    FARGATE_RAM_COST = 0.004445 / 60 / 60
    if previous_result and 'cost_usd' in previous_result:
        cpu_cost = previous_result['cost_cpu'] + FARGATE_CPU_COST
        ram_cost = previous_result['cost_ram'] + (ram['used']/1024/1024/1024) * FARGATE_RAM_COST
    else:
        cpu_cost = FARGATE_CPU_COST
        ram_cost = (ram['used']/1024/1024/1024) * FARGATE_RAM_COST

    return { 'cost_cpu': cpu_cost, 'cost_ram': ram_cost, 'cost_usd': ram_cost + cpu_cost }
```

Figure 8: Example cost function used in this thesis

By following the example described here one can define different cost functions based on their cost structure. As already mentioned most often costs will be a composition of resources and time so one should be able to define a higher metric just based on these metrics. The key is that the developer has to know the cost structure to define a usable cost function.

6.2.3 Pushing custom metrics

Typically tasks built with the scripts provided by the system will have a *code.py* file in the root. This file will already contain a line that initializes the performance monitor. This can be seen in Figure 9.

To push a metric to the message queue the user has to just call a function called `log_analytics_metric` on the performance monitor instance and provide the name of the metric as the key and the value as the value inside of a dictionary. An example push of a custom metric can be seen in Figure 10.

```python
def main():
    params = utils.get_args_params(sys.argv)
    pm = PerformanceMonitor(task_name='score_linear_regression', pipeline_id=params['pipeline_id'])
```

Figure 9: Initial state of the main function in code.py

```python
# model_score returns a dict -> { 'r2_squared': r2_squared_score }
model_score = score_model(store, model, data_path, preset)
pm.log_analytics_metric(model_score)
```

Figure 10: Logging custom metrics

This function would be usually used to define and store metrics related to the quality of the analysis. An example metric would be the `R` squared value. The users are responsible to retrieve the correct value themselves and defining the pushing of that value. After that, the value will be used together with other metrics by the control consumer to manage quality of analytics.

6.2.4 Setting custom control rules

The control consumer is a separate service which is built from a Docker container. At the root of that container it holds the *code.py* file that can be viewed in this path from the repository root - `/containers/control/code.py`. This file initializes the consumer service object. By default, it is initialized without any arguments meaning that there are no control rules set. To set up control rules we have to pass a python function to the initialization call. The function receives the message retrieved from the RabbitMQ queue and can stop or pause the pipeline depending on a value in that message body

Typically there are messages from three different indexes that the function can receive. The `data_logs` index contains information about the incoming and outgoing data sizes and the name of the file. The `analytics` index contains the previously defined custom metrics for analytics, for example the `R` squared value. The `metrics` index contains the raw resource metrics, which are `cpu`, `ram` and `time_elapsed` as well as the result of the previously defined cost function.

To pause or stop the pipeline the function has to return either the string `HARD_STOP` for stopping of `SOFT_STOP` for pausing. The function has access to all the previously mentioned indexes. The user is in charge of defining the rules based on the values from these indexes. The consumer will interact with the running pipeline based on the return value of the function. An example function can be seen in Figure 11.

6.2.5 Changing the storage

The storage is also defined as a python class and it basically implements methods for file downloading and uploading and makes sure that the basic information regarding

```python
def default_get_control_action(body_dict):
    index = body_dict.pop('metric_type', None)
    print(body_dict, flush=True)
    try:
        if index == 'metrics':
            if body_dict['cost_usd'] > 1 or body_dict['time_elapsed'] > 500:
                return 'SOFT_STOP'
            elif body_dict['time_elapsed'] > 1000:
                return 'HARD_STOP'

        elif index == 'data_logs':
            if body_dict['task_name'] == 'clean_data':
                if body_dict['in']['train.csv'] / 2 > body_dict['out']['train.csv']:
                    return 'SOFT_STOP'

        elif index == 'analytics':
            if body_dict['payload']['r2_squared'] < 0.2:
                return 'SOFT_STOP'
        else:
            print('No valid index found!')
            return -1
    except KeyError:
        pass
```

Figure 11: Example control function used in this thesis

these processes is logged to the performance monitor instance.

The storage is defined in the base container root in the *store.py* file that can be viewed in this path from the repository root - `/containers/base/store.py`. It can be customized by defining a different storage solution and initializing it instead. The minimum requirements for a custom storage class is that it has to have 2 functions - `save_file` and `get_file` and one class variable - `performance_monitor` which is always set at the start of a task. The information about the data can then be logged by using the `log_infile` and `log_outfile` functions that both take the local path to the file and extract information automatically.

6.3 Summary

The high-level architecture discussed in the section allows us to set up pipelines with modular tasks that can be swapped and re-used across different pipelines. It is designed to be cloud-agnostic. It features a component for storing and visualizing metrics and also includes a solution for serving models. These things directly satisfy the first, second, third and fifth requirement from section 2.3.

The fourth requirement from section 2.3. is satisfied by generalizing the design and allowing custom functions for retrieving metrics and controlling the process as discussed in the previous subsection. The trade-off of generalizing the system and making it usable for different domains is that the end-user requires to put in more work. The biggest advantage of this design is that it allows us to retrieve task-specific metrics for resource consumption which is a feature that is not supported

by most machine learning systems. In general, this approach allows the user to be in control over data pipeline management with respect to quality of analytics.

7 Framework evaluation

To determine which are the best strategies for managing quality of analytics and estimate the overall overhead cost of using such a framework we ran tests on the pipeline in different setups. This evaluation should show better strategies for QoA management in retail forecasting and highlight the strengths and weaknesses of the designed system. This section describes the tests that were performed and their results.

Before going into the results we describe the test scenarios, environment, and relevant metrics. This then provides a better context for the results. Together we ran 3 different pipelines, each with 4 different datasets and all of them used linear regression as the training algorithm. Linear regression was selected as it is a more simple algorithm that allows to focus on the data engineering aspects rather and leave more complex data science aspects out of the scope of this thesis. The task of the linear regression was to create a model that can predict future sales of an item at a specific location.

7.1 Description of data

As mentioned before for testing we used 4 datasets of different size. The row count for the datasets were 10 000, 100 000, 500 000, 1 000 000 rows respectively. Each of the datasets was generated using the smallest 10 000 row dataset as the base so the general characteristics for all of them were the same.

The base dataset contains sales data. It was created in a way that mimics actual datasets from retailers. This dataset was selected as the thesis case is set in retail. We cannot use real retail datasets not to leak any sensitive information. The setup that was achieved is close to the one used in retail analytics and thus the results from this evaluation would be applied.

The datasets contained information about item promotions and sales in each store every day over 3 years. Five sample rows from the dataset can be seen in Figure 12.

date	item_id	item_name	volume	unit_price	unit_cost	promo_	category_ne	category_margin	category1	category2	location	sales
07/01/2018	100	Chicken	38144.0	3.79	2.7	0	451692.0	0.25	Meat	Food	Helsinki	144565.76
14/01/2018	100	Chicken	36420.0	3.79	2.66	0	414342.0	0.25	Meat	Food	Helsinki	138031.8
21/01/2018	100	Chicken	35322.0	3.79	2.66	0	381854.0	0.25	Meat	Food	Helsinki	133870.38

Figure 12: Sample rows from the dataset

7.2 Description of QoA management strategies

To understand how one can better manage the trade-offs between data quality, time, cost and other related metrics we defined a set of trade-off management strategies. These strategies are defined based on the research done previously in this thesis.

The strategies were empirically deployed and are evaluated in further sections of this thesis.

The first subset of strategies manages resource consumption directly by utilizing two different environments that consume different amounts of computational resources [52]. One environment would be more optimized towards cost while the other one towards time. The simplest way to do this is to first have a cloud computing resource with very high capacity which yields to higher costs and second to have a different cloud computing resource with a limited capacity which yields lower costs but requires more time to execute a task. In practice, the difference lies in the memory size of instances used for computation since the data analysis tasks require easily accessible storage for swaps and computations.

The second set of management strategies focus on changing the data analytics process directly. The most basic idea would be to train the models on different levels of granularity. One example would be training a sales prediction model only based on the sale date and category of the item, meaning that we disregard the item identifier or location. This approach would most likely prove ineffective as the retail market is highly competitive and requires the highest quality possible. One example of such an ineffective result would be making a prediction on how a specific soft drink performs based on the historic performance of all soft drinks. It is very unlikely that such a level of granularity would prove useful in a real-life scenario as items within a product category vary and taking an average for each item from this level would be a poor approximation in most cases. So instead the focus is on decreasing the size of the result set.

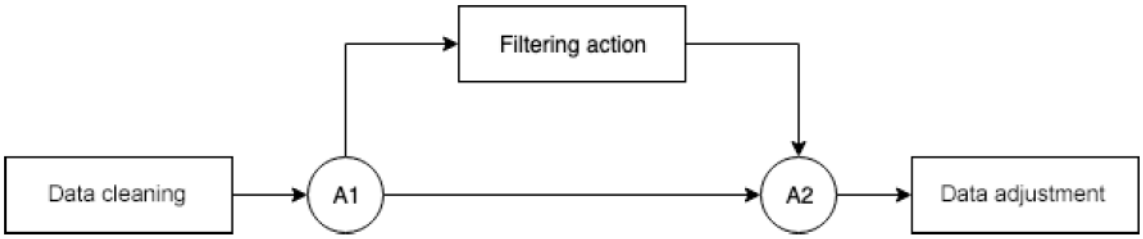

Figure 13: PAM control action schema

The result of forecasting pipelines in retail typically is a set of models that can be queried to make predictions about a specific location and item. This means that different models are used for each location and item pair. So to still maintain the best possible quality per model we can reduce the number of models. This is done by filtering out certain locations, items or time periods. The filtering would be done by a primitive action model (PAM) control action that based on the retrieved metrics applies filtering to the dataset after each task within a pipeline [11]. A high-level model of this approach can be seen in Figure 13. Where the intermediate monitoring action A1 receives task-level metrics about resources and data quality in real-time and depending on these metrics decides to either filter the data or proceed without filtering.

7.3 Description of metrics

During the evaluation, we set up the framework to observe four key metrics.

The primary quality metric that was observed was row count. The filtering actions reduce the number of rows used for training or handling the data. So the general quality metric in this setting can be simplified as rows used for training. The reduction of rows yields lower cost and reduces time with a price of decreased quality. It is useful to point out that filtering the data too much could cause more overall noise and negatively impact the quality of the model. So the general quality metric in this setting can be simplified as rows used for training.

A secondary quality metric that was observed was R squared value. This is a good metric since we were looking at linear regression algorithms. An acceptable R squared value largely depends on the use case and task at hand. The guidelines that will be used to evaluate the r squared value are taken from Ref [53] that looks at quality metrics in marketing research. From Ref [53] "As a guideline, R squared values of 0.75, 0.50 and 0.25 can be considered substantial, moderate and weak." These values are the ones that will be used as indicators for quality depending on the R squared score.

Additionally, we had two resource-related metrics - elapsed time and cost. Elapsed time was expressed in seconds and calculated on a task per task basis. The Fargate cost function was used for determining the running cost of a task. This cost function is expressed in Python in Figure 8.

7.4 Description of pipelines

As mentioned before we ran three different pipelines. One pipeline did not utilize the performance monitor and control management services at all and this pipeline was created to determine a baseline for startup time and base task execution times. This base pipeline consisted of four different tasks. First data cleaning that removes faulty rows and calculates the sales for each item from raw data. Second, a data transformation step that formats the different categorical values so that they can be used in the algorithm. Third a linear regression step that runs over the data and creates a prediction model. And fourth a scoring task that scores the resulting model and expresses the quality as the previously mentioned R squared score.

After that, we built a pipeline that adjusts the data based on different metrics to control the cost and elapsed time of a pipeline. For this pipeline, we created two different adjustment actions, one of them could be performed after the data cleaning task and the other one after the data transformation task. The first data adjustment action reduced the number of store locations that are taken into account when training the model. Our dataset contained information about stores from two different cities, so the data adjustment action typically would filter the results to one city effectively decreasing the row count 2 times. The second data adjustment action filtered out rows so that only information about the last available 12 months is present. This effectively removed 24 additional months of data and decreased the row count 3 times. So in general if both of the actions ran on the dataset the total

row count would be decreased by around 6 times. The pipeline in airflow can be seen on the left-hand side in Figure A.1 in the Appendix section.

The third pipeline that was built was controlling quality of analytics by provisioning different instances to the tasks. Again we had different branches for the tasks. We ran the data transformation task on two different instances, one with 4GB of RAM available and 2VCPU and another with 1GB of RAM available and 1VCPU. The same instances were available for the linear regression task that trains the model. The pipeline in airflow can be seen in on the right-hand side in Figure A.1 in the Appendix section.

7.5 Adjustment action strategy evaluation

The first strategy that was applied for managing the mentioned trade-offs was adding additional adjustment actions in the pipeline. The goal of these actions is to reduce the row count used for training that directly impacts quality of analytics metrics.

The results for the two smaller datasets were not acceptable. For the smaller datasets, the cost and time overhead caused by the adjustment actions would make the total cost of a pipeline greater than a pipeline without these actions. This means that for smaller datasets this strategy would produce worse results but increase the total cost and elapsed time.

The results were better for the two bigger datasets. For the dataset with 500 000 rows, the total cost for the pipeline with adjustment actions was the same as for the regular pipeline without these actions. The quality of the result was worse and the time spent on computation was also higher. These differences are reflected in Figure 14, where the top image shows the gain in cost and the bottom image shows the gain in elapsed time. In both cases, smaller is better.

A real improvement for this strategy could be seen for the biggest dataset with 1 million rows. The total cost was 13,3% lower and the elapsed time was 44% shorter. This improvement came with a cost towards the quality of results. The R squared value was 9,5% lower at 77% and the row count was 6 times smaller. This means that with a lowered cost we could get good enough results for 50% of the total store locations. The trade-off between the various metrics is plotted in Figure 15, where the top image shows the gain in cost and the bottom image shows the gain in elapsed time. Again in both cases, smaller is better.

To better understand the dynamics of this strategy we plotted the task level cost for the pipeline with adjustment actions next to the task level cost for the pipeline without adjustment actions. This comparison can be seen in Figure A.2 in the Appendix section. The adjustment actions are colored in gray. In this figure, we can observe that the first adjustment action reduces the cost of the next task for about two times if we compare to the pipeline without adjustments. The second adjustment action reduces the cost of the next task by about three times if we compare to the pipeline without adjustments.

Overall this strategy proved effective and could be applied in practice in retail analytics as the cost and time improvement is significant. A typical use case for applying this strategy would be getting sample results for new clients.

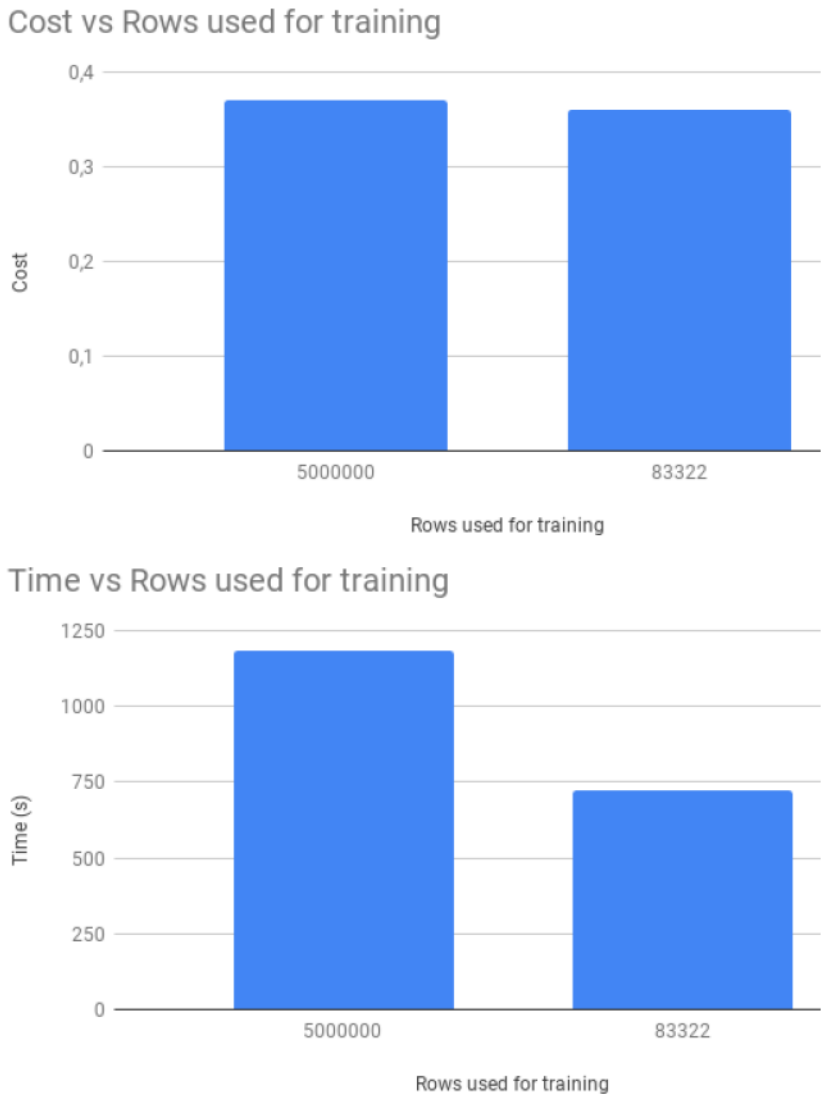

Figure 14: Trade off comparison for 500 000 rows

7.6 Resource control strategy evaluation

Another strategy that was applied was controlling the cost and time by allocating different resources. For this purpose, different docker images were built with different resources and then tested locally. A similar approach could be done by deploying the containers to differently sized cloud server instances. And running the appropriate instance on demand.

With this strategy, we did not manage to achieve a meaningful difference in cost. This is because the tasks that utilized fewer resources took more time and essentially the cost was the same since the cost is a function of time. The difference in time can be seen in Figure A.3 in the Appendix section. On the top of that figure, we can observe the time elapsed per task for a pipeline without adjustments and on the bottom, we can observe the time elapsed per task for a pipeline with adjustments.

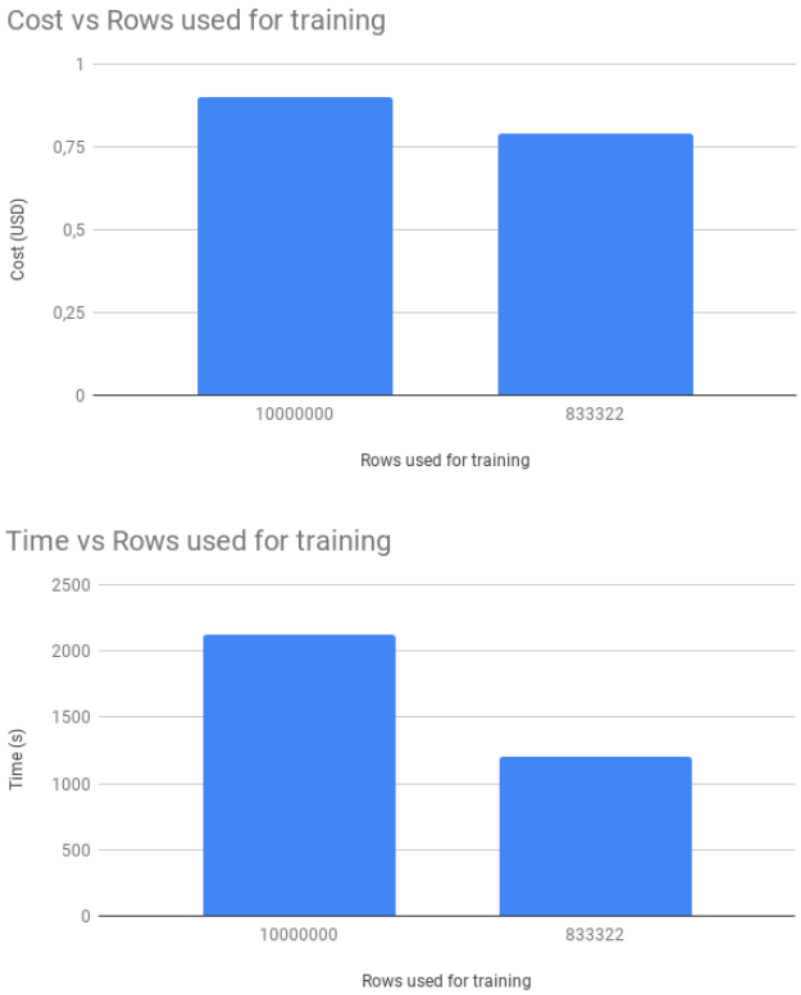

Figure 15: Trade off comparison for 1 000 000 rows

In both cases, the quality remains the same.

7.7 Auxillary evaluation results

The monitoring tools enabled us to observe an interesting issue with the pipelines. For some tasks, most of the total time spent on a task when dealing with large datasets is spent on getting the data into local memory. This means downloading the dataset and storing it locally for access. This proportion is represented in Figure 16. In which we can see that 77% of the total task time is spent on downloading the dataset. This issue should be addressed in the future and the situation can be improved by using different storage solutions.

Another observation made from multiple runs of the framework is that the startup can take a significant amount of time. Especially when the dataset is small. The service that requires the longest time to fully initialize is ElasticSearch since it is also dependent on other services in the system. To measure the startup time

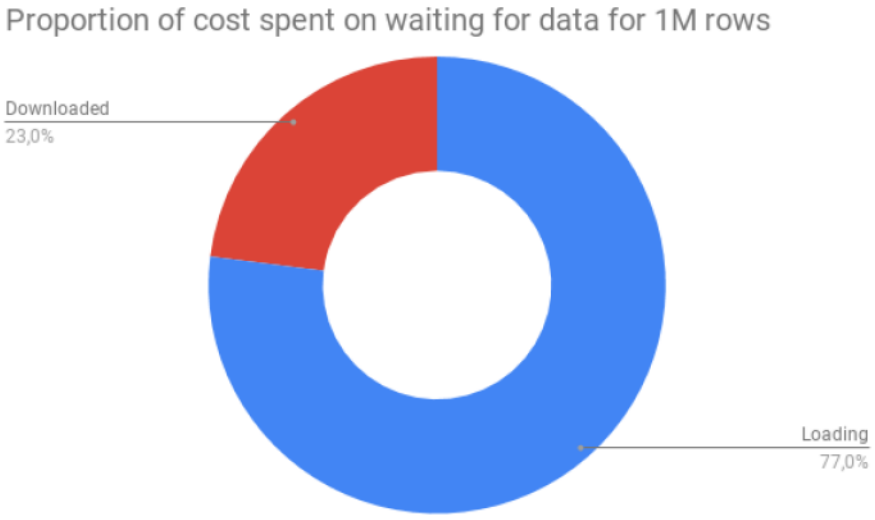

Figure 16: Pipeline cost with adjustment actions vs without adjustment actions

the framework was started in two different environments. The first environment had 2vCPUs and 5GB of RAM. The second environment had 3vCPUs and 7GB of RAM. In both cases, the overall startup time averaged at around 3 minutes. The full breakdown of these results can be seen in Table 2.

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Environment 1	210s	190s	197s	198s	211s	201,2s
Environment 2	170s	177s	180s	166s	180s	174,6s

Table 2: Framework startup times

7.8 Summary

To summarize the best strategy for the test retail datasets was to perform adjustment actions on the data that reduce the count of rows by filtering by geographical location or date. The effectiveness of this strategy increases with the increase in data size. These results show that QoA management can be fruitful for offline learning cases in retail analytics. The advantages for QoA in these cases is seen when utilizing big datasets and when optimizing the trade-off between cost and quality with a strategy that adjusts data to obtain the target cost. There still are ways to improve the system and they will be discussed in-depth in the next section.

8 Discussion

This section contains a critical view of the designed framework. The first subsection asseses if the framework fulfills the set out requirements. The second subsection looks into possible future work regarding the framework.

8.1 Meeting the requirements

The set of system requirements were listed in section 2.3. It is now time to revisit them to determine to what extent these requirements were satisfied.

The first requirement was that the system has to be cloud-agnostic. This was done to a certain extent. The QoA components of the framework are built on top of open-source software so they can be deployed and re-used across different environments. The module that is currently tied to a specific cloud provider is the storage module. While currently, the framework has only support for AWS S3 storage it can be extended to work with different storage solutions.

The second requirement was that the task structure within pipelines has to be modular. This requirement was targeted at the pipeline level of the system but the selected task design was influenced with QoA support in mind. The tasks and the QoA components were containerized. This approach brings a certain level of modularity as the containers can be reused across different pipelines with the build pipeline infrastructure.

The third level was task level visual feedback about ongoing processes within pipelines. This requirement was fully satisfied with the introduction of the Kibana UI component. While this component does not impact the QoA management process directly it allows making better decisions when applying adjustment actions.

The fourth requirement is directly linked with QoA. This requirement stated that an interface must be provided that can be used to interact with the pipelines to control the processes with respect to quality of analytics. This requirement was directly satisfied by the control consumer which interacts with the pipelines based on a predefined set of control rules.

The fifth requirement was specific to the end system and the thesis case and it stated that the API which exposes the models has to be restricted from public access. This requirement was satisfied by using AWS Lambda functions which expose the models to an HTTP endpoint. This endpoint only accepts requests from a predefined range of IP addresses. This restriction was made with the help of AWS Virtual Private Networks that contain the Lambda functions. To provide an additional layer of security all the requests can be proxied through a backend service that manages authentication for the user interface. Then the AWS Lambda function would only need to accept requests from this one peer. Which is a simple adjustment if the backend services also lies in the same VPC as the AWS Lambda endpoint.

To summarize all the requirements were largely met. The system could still be improved to be more modular by making the QoA components and storage environment agnostic so that they can be re-used without any additional setup in data pipelines that utilize different technologies.

8.2 Future work

This subsection highlights the parts of the system that should be improved in the future.

First, let us look at the system from a high level. In the future, the framework should be modified to improve startup and data retrieval times. To achieve a quicker startup time we should allow ElasticSearch related services to load asynchronously by locally caching the messages from the queue which are not picked up by ElasticSearch and then consuming them at a different point. Currently, the consumer does not retry sending the data if ElasticSearch is unresponsive or returns an error. Improvement of data retrieval times is not as trivial and would require deeper research of existing solutions. To find the best solution one should run empirical tests of different data storage solutions and see which ones can provide improvement.

More interesting research challenges for the future are associated with the automation of the trade-off management process. Currently, this process is done manually by selecting the next appropriate branch depending on a set of metrics. In an ideal environment, the framework could support various analytics algorithms with little or no actions required from the data scientist. The system should be able to apply an appropriate adjustment action in real-time based on the consumed metrics about resources and quality. The appropriate adjustment action means that the system maximizes the quality given the maximum time and resource or minimizes the time and resource given the minimum acceptable quality. One interesting strategy that could be studied further is building different analytics algorithms that interact with the same data and have the adjustment action trigger a different algorithm that is more likely to achieve the expected trade-off based on the received metrics. This is both an engineering and learning problem that requires close collaboration between experts of both fields.

Another interesting challenge would be testing the framework in an online training environment that supports on-demand quality of analytics. An example of on-demand quality of analytics would be a courier requesting the best possible path for delivering a package given limited time and resources. Such a setting gives direct access to quality of analytics for an important set of shareholders - the end-users. This would mean that not only has the framework be able to provide optimal decision making within a pipeline based on streamed metrics but also provide a user interface for the end-user who is not familiar with data science or software engineering. The simplest version of this user interface could allow to define bounds for time and cost and execute pipelines that take these bounds into account.

9 Conclusion

In this thesis, we looked at quality of analytics management in an offline learning setting for retail. The thesis describes quality of analytics and reasons about the strategies associated with its management. A thorough comparison of the state of the art machine learning systems and related technologies was conducted. This comparison revealed that current machine learning systems do not fully support quality of analytics management. Based on the requirements and gathered best practices we designed a suitable architecture for the solution. The final solution used Apache Airflow as the foundation for pipelines. Individual tasks were wrapped in Docker containers. Wrapping the task in a container gives modularity and allows to scale each task separately. Each of the containers is set up with a performance monitoring package that the user can use to capture metrics about the running processes in real-time. These metrics are consumed by two consumers. One of them manages the metric transfer to ElasticSearch. The other consumer is the key element for having control over the processes as it can pause or stop the pipeline according to a predefined set of rules. The framework uses the current state of the art technologies and employs an event-based architecture for monitoring and controlling the processes. The proof of concept was open-sourced and is available at https://github.com/kristsellforte/qoa_framework for further development. With the built setup the data scientist can interact with running pipelines and visualize the running processes. The solution was tested in different environments with retail datasets of different sizes.

From the theoretical results, we can conclude that currently, various strategies exist for managing quality of analytics in data pipelines. The most popular strategies would be adjusting the dataset with the help of primitive action models or limiting the available resources on the task level. Different companies, settings, and analytics processes have different needs and the most useful strategy heavily depends on those needs.

The main practical results are the following. Firstly, using adjustment actions that are contained as primitive action models is a good strategy for managing the result in retail data pipelines with respect to quality of analytics. Secondly, the advantages of quality of analytics management increase with larger datasets. Thirdly, controlling available resources for separate tasks did not show helpful for managing the trade-off between cost and quality but this approach should be revisited and evaluated again for different settings. Finally, the introduction of additional services and tasks that handle quality of analytics management comes with an overhead cost that can be offset when utilizing the framework with larger datasets.

Generally, quality of analytics management can be useful for offline learning cases in retail analytics. While the savings by employing these processes only measure in USD cents when putting into perspective that these pipelines are run multiple times with different datasets the total savings increase by up to 100 times. Currently, the framework still can be improved but it can already provide a better understanding of the cost structure for data scientists and it provides a ground for experimenting with different setups that manage different trade-offs.

The trade-off management should be studied further and this thesis only covers the first step to fully automated quality-aware data pipelines. Quality metrics often depend on the underlying analytics processes and are different for different cases. The data scientist is in charge of defining those metrics and proposing rules that control the result with respect to quality of analytics. Full automation of these processes is challenging and to achieve this tight collaboration between the fields of data science and software engineering is required.

References

[1] Deloitte. Machine learning: things are getting intense. Accessed 21.07.2019. Accessible at: https://www2.deloitte.com/content/dam/Deloitte/global/Images/infographics/technologymediatelecommunications/gx-deloitte-tmt-2018-intense-machine-learning-report.pdf

[2] Weber, B. G. (2018) *Data science for startups*

[3] Ben-David, Shai and Kushilevitz, Eyal and Mansour, Yishay *Online Learning versus Offline Learning* Accessed 21.07.2019. Accessible at: https://doi.org/10.1023/A:1007465907571

[4] Sellforte. Sellforte website. Accessed 21.07.2019. Accessible at: https://www.sellforte.com/

[5] Polyzotis, Neoklis and Roy, Sudip and Whang, Steven Euijong and Zinkevich, Martin *Data Management Challenges in Production Machine Learning* Accessed 14.03.2019. Accessible at: http://doi.acm.org/10.1145/3035918.3054782

[6] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, Gyuri Szarvas *On Challenges in Machine Learning Model Management* Accessed 14.03.2019. Accessible at: http://sites.computer.org/debull/A18dec/p5.pdf

[7] Ulbricht, Robert and Donker, Hilko and Hartmann, Claudio and Hahmann, Martin and Lehner, Wolfgang *Challenges for Context-Driven Time Series Forecasting* Accessed 15.03.2019. Accessible at: http://doi.acm.org/10.1145/2896822

[8] van der Weide, Tom and Papadopoulos, Dimitris and Smirnov, Oleg and Zielinski, Michal and van Kasteren, Tim *Versioning for End-to-End Machine Learning Pipelines* Accessed 14.03.2019. Accessible at: http://doi.acm.org/10.1145/3076246.3076248

[9] Truong, Hong-Linh and Murguzur, Aitor and Yang, Erica *Challenges in Enabling Quality of Analytics in the Cloud* Accessed 15.03.2019. Accessible at: https://dl.acm.org/citation.cfm?id=3138806&dl=ACM&coll=DL

[10] Hazen, Benjamin and Boone, Christopher and Jones-Farmer, L.A. and Ezell, Jeremy *Data Quality for Data Science, Predictive Analytics, and Big Data in Supply Chain Management: An Introduction to the Problem and Suggestions for Research and Applications*

[11] Nguyen, Tien-Dung and Truong, Hong-Linh and Copil, Georgiana and Le, Duc-Hung and Moldovan, Daniel and Dustdar, Schahram *On Developing and Operating of Data Elasticity Management Process*

[12] N. Harth and C. Anagnostopoulos *Quality-aware aggregation and predictive analytics at the edge*

[13] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu *Edge Computing: Vision and Challenges* Accessed 21.07.2019. Accessible at: https://ieeexplore-ieee-org.libproxy.aalto.fi/document/7488250

[14] Amazon Web Services. AWS Sagemaker. Accessed 12.03.2019. Accessible at: https://aws.amazon.com/sagemaker

[15] Bilgorov, A. *Building fully custom machine learning models on AWS SageMaker: a practical guide* Accessed 12.03.2019. Accessible at: http://bitly.com/2UZkVEp

[16] Amazon Web Services. AWS Lambda. Accessed 24.07.2019. Accessible at: https://aws.amazon.com/lambda/

[17] Microsoft Azure. Azure Machine Learning Documentation. Accessed 12.03.2019. Accessible at: https://docs.microsoft.com/en-us/azure/machine-learning/

[18] Google Cloud. Cloud Machine Learning Engine. Accessed 12.03.2019. Accessible at: https://cloud.google.com/ml-engine/

[19] Pachyderm. Pachyderm Developer Documentation. Accessed 12.03.2019. Accessible at: http://docs.pachyderm.io

[20] Pachyderm. Pachyderm Github Repository. Accessed 12.03.2019. Accessible at: https://github.com/pachyderm/pachyderm

[21] Apache PredictionIO. Apache PredictionIO Documentation. Accessed 12.03.2019. Accessible at: https://predictionio.apache.org/

[22] Valohai. Valohai Documentation. Accessed 12.03.2019. Accessible at: https://docs.valohai.com/

[23] Kubeflow. Kubeflow documentation. Accessed 12.03.2019. Accessible at: https://www.kubeflow.org/docs

[24] Amazon Web Services. Pricing. Accessed 12.03.2019. Accessible at: https://aws.amazon.com/pricing/

[25] DVC *Comparison to Existing Technologies* Accessed 09.04.2019. Accessible at: https://dvc.org/doc/dvc-philosophy/related-technologies

[26] Polyaxon *Polyaxon documentation* Accessed 30.06.2019. Accessible at: https://docs.polyaxon.com/

[27] Amazon Web Services. Step functions. Accessed 20.03.2019. Accessible at: https://aws.amazon.com/step-functions/

[28] Luigi. Luigi repository. Accessed 20.03.2019. Accessible at: https://github.com/spotify/luigi

[29] Netflix. Netflix Conductor Documentation. Accessed 20.03.2019. Accessible at: https://netflix.github.io/conductor/

[30] Apache. Apache Airflow Documentation. Accessed 20.03.2019. Accessible at: https://airflow.apache.org/

[31] Podeswa, Yasha *Let's build a Service Oriented Data Pipeline* Accessed 21.03.2019. Accessible at: https://www.youtube.com/watch?v=rl5CoonAYB8

[32] Kaniovskyi, Yuriy and Koehler, Martin and Benkner, Siegfried *A Containerized Analytics Framework for Data and Compute-intensive Pipeline Applications* Accessed 20.03.2019. Accessible at: https://airflow.apache.org/

[33] Amazon Web Services. Amazon S3. Accessed 21.07.2019. Accessible at: https://aws.amazon.com/s3/

[34] Y. Moatti and E. Rom and R. Gracia-Tinedo and D. Naor and D. Chen and J. Sampe and M. Sanchez-Artigas and P. Garcıa-Lopez and F. Gluszak and E. Deschdt and F. Pace and D. Venzano and P. Michiardi *Too Big to Eat: Boosting Analytics Data Ingestion from Object Stores with Scoop*

[35] Pallets Projects. Flask. Accessed 21.07.2019. Accessible at: https://palletsprojects.com/p/flask/

[36] Amazon Web Services. AWS Service Limits. Accessed 19.03.2019. Accessible at: https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

[37] Tensorflow. Install TensorFlow. Accessed 21.07.2019. Accessible at: https://www.tensorflow.org/install

[38] Facebook. Prophet Installation. Accessed 21.07.2019. Accessible at: https://facebook.github.io/prophet/docs/installation.html

[39] K. Kritikos and P. Skrzypek *A Review of Serverless Frameworks*

[40] Google cadvisor. Cadvisor repository and documentation. Accessed 30.06.2019. Accessible at: https://github.com/google/cadvisor

[41] Docker CLI. Docker stats command. Accessed 30.06.2019. Accessible at: https://docs.docker.com/engine/reference/commandline/stats/

[42] Psutil. Documentation. Accessed 30.06.2019. Accessible at: https://psutil.readthedocs.io/en/latest/

[43] RabbitMQ. RabbitMQ Website. Accessed 24.07.2019. Accessible at: https://www.rabbitmq.com/

[44] Elastic. ElasticSearch product. Accessed 24.07.2019. Accessible at: https://www.elastic.co/products/elasticsearch

[45] Elastic. Kibana Product. Accessed 24.07.2019. Accessible at: https://www.elastic.co/products/kibana

[46] Apache Airflow *Concepts* Accessed 15.05.2019. Accessible at: https://airflow.apache.org/concepts.html

[47] Drivy *Airflow Architecture at Drivy* Accessed 15.05.2019. Accessible at: https://drivy.engineering/airflow-architecture/

[48] M. Villamizar and O. Garcés and L. Ochoa and H. Castro and L. Salamanca and M. Verano and R. Casallas and S. Gil and C. Valencia and A. Zambrano and M. Lang *Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures* Accessed 19.03.2019. Accessible at: https://ieeexplore.ieee.org/document/7515686

[49] Jonas, Eric and Pu, Qifan and Venkataraman, Shivaram and Stoica, Ion and Recht, Benjamin *Occupy the Cloud: Distributed Computing for the 99%* Accessed 19.03.2019. Accessible at: http://doi.acm.org/10.1145/3127479.3128601

[50] AWS *ECS Fargate pricing* Accessed 31.05.2019. Accessible at: https://aws.amazon.com/fargate/pricing/

[51] S. Nixon *What's in a vCPU: State of Amazon EC2 in 2018* Accessed 31.05.2019. Accessible at: https://www.credera.com/blog/technology-solutions/whats-in-a-vcpu-state-of-amazon-ec2-in-2018/

[52] Garcia, Abel and Laneve, Cosimo and Lienhardt, Michael *Static Analysis of Cloud Elasticity* Accessed 15.03.2019. Accessible at: http://doi.acm.org/10.1145/2790449.2790524

[53] Joseph F., Jr, Hair, and Ringle, Christian and Sarstedt, Marko *PLS-sem: Indeed a silver bullet*

Appendix

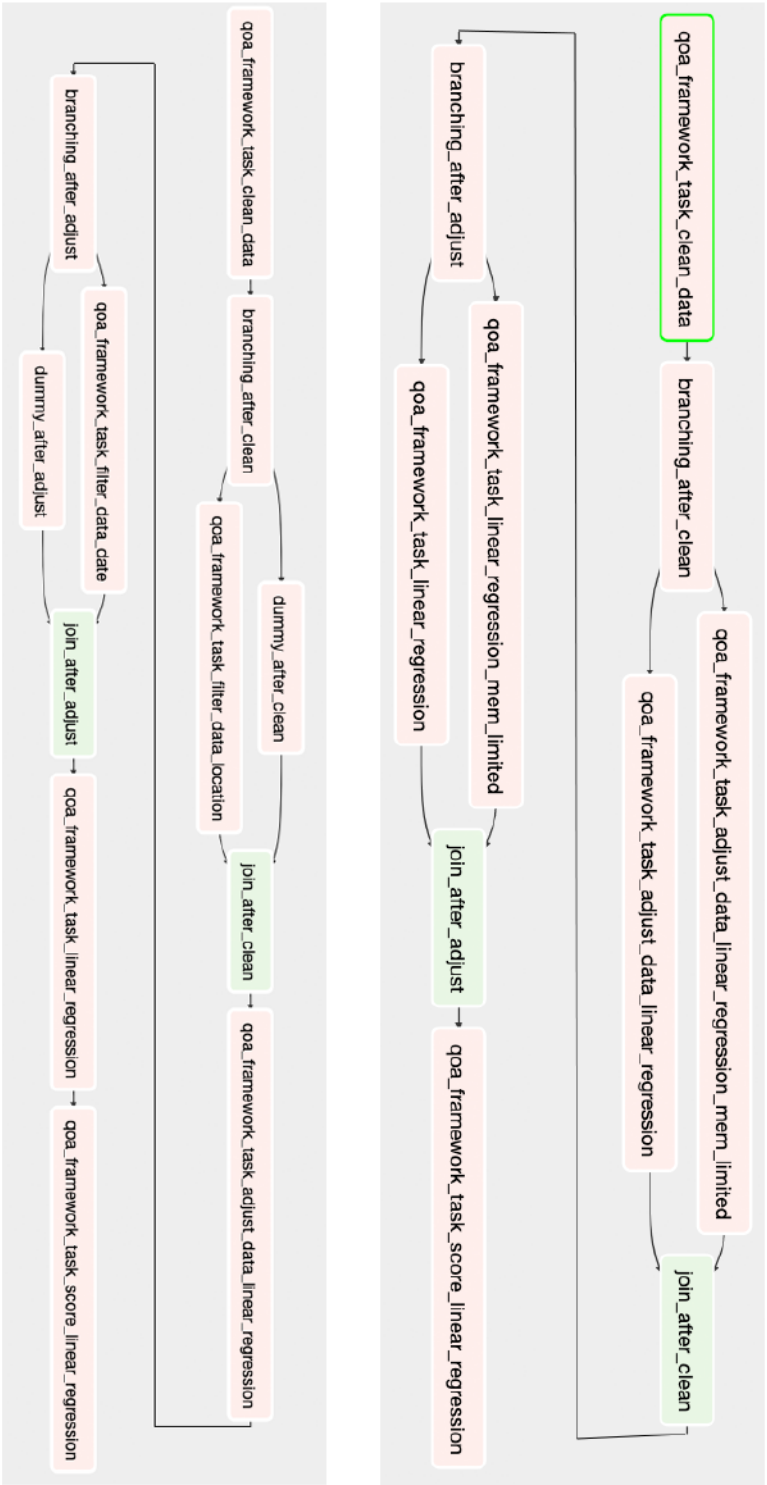

Figure A.1: Pipelines used for evaluation

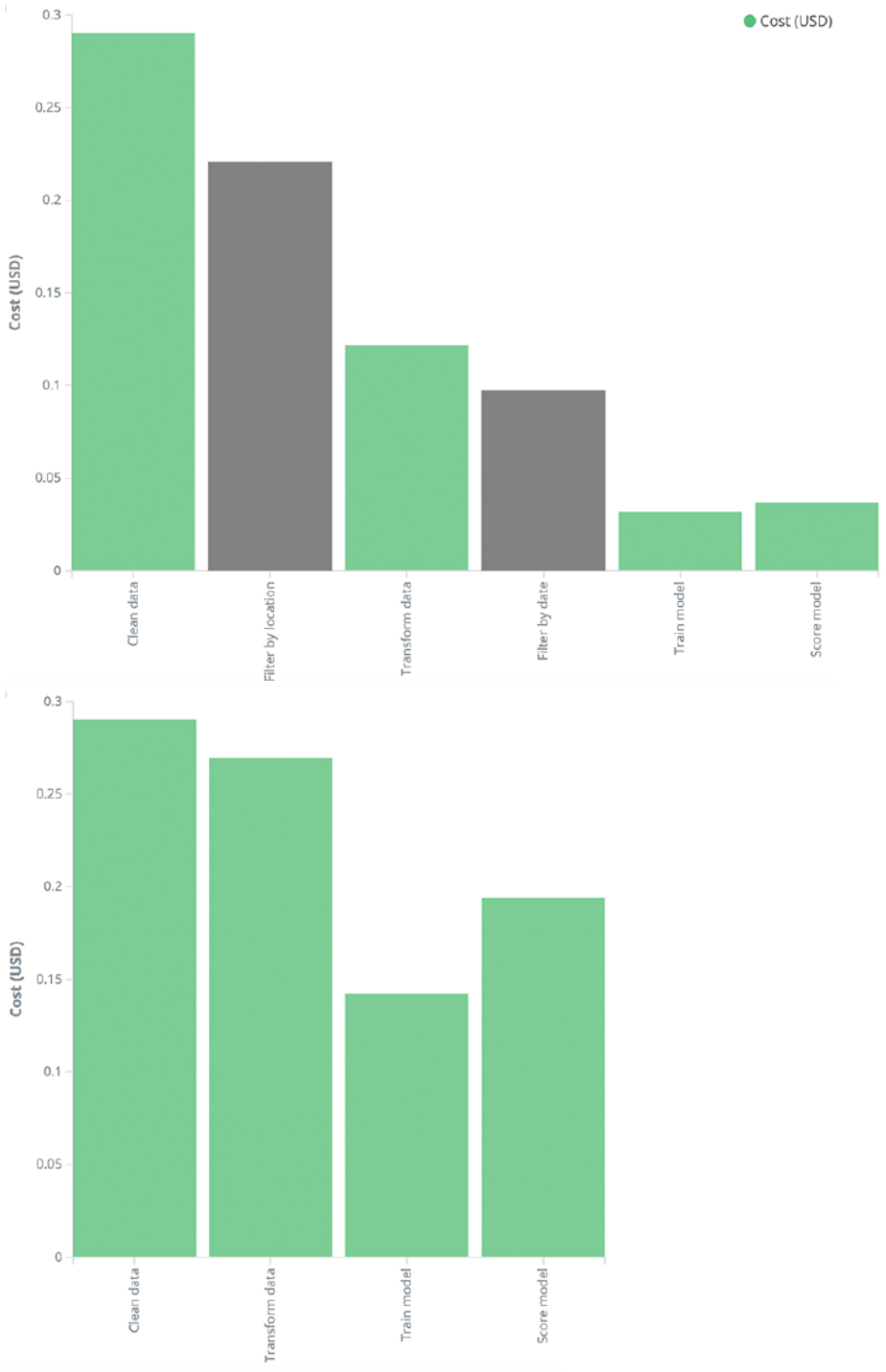

Figure A.2: Pipeline cost with adjustment actions vs without adjustment actions

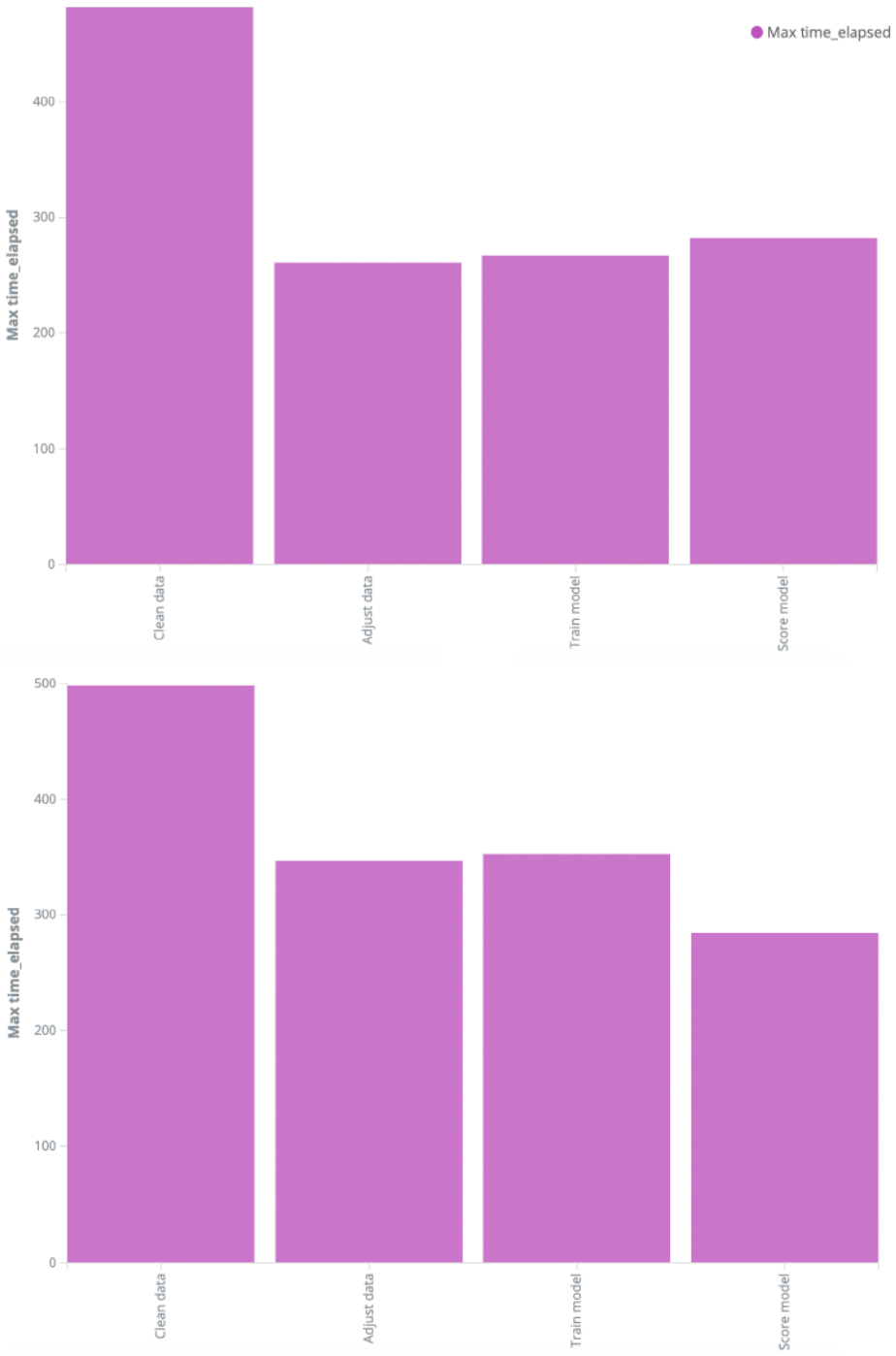

Figure A.3: Pipeline cost with adjustment actions vs without adjustment actions